Data Model for Document Transformation and Assembly (Extended Abstract)

Makoto Murata[1]

Fuji Xerox Information Systems Co., Ltd., KSP 9A7, 2-1 Sakado 3-chome, Takatsu-ku, Kawasaki-shi, Kanagawa-ken, Japan 213
murata@fxis.fujixerox.co.jp

Abstract. This paper shows a data model for transforming and assembling document information such as SGML or XML documents. The biggest advantage over other data models is that this data model simultaneously provides (1) powerful patterns and contextual conditions, and (2) schema transformation. Patterns and contextual conditions capture conditions on subordinates and those on superiors, siblings, subordinates of siblings, etc, respectively, and have been recognized as highly important mechanisms for identifying document components in the document processing community. Meanwhile, schema transformation has been, since the RDB, recognized as crucial in the database community. However, no data models have provided all three of patterns, contextual conditions, and schema transformation.

This data model is based on the forest-regular language theory. A schema is a forest automaton and an instance is a finite set of forests (sequences of trees). Since the parse tree set of an extended-context free grammar is accepted by a forest automaton, this model is a generalization of Gonnet and Tompa's grammatical model. Patterns are captured as forest automatons; contextual conditions are pointed forest representations (a variation of Podelski's pointed tree representations). Controlled by patterns and contextual conditions, an operator creates an instance from an input instance and also creates a reasonably small schema from an input schema. Furthermore, the created schema is often *minimally sufficient*; any forest permitted by it may be generated by some input instance.

1 Introduction

Document information, such as SGML or XML documents, is typically organized for particular purposes. However, the same information can be, if reorganized, utilized for different purposes. For example, Northwest may want to reorganize document information provided by Boeing and McDonnell-Douglas so as to efficiently maintain their airplanes in their own manner. Transformation and assembly of SGML/XML documents are expected to provide such reorganization, and have attracted strong interest in the SGML/XML community.

It is an exciting challenge to provide a data model for such transformation and assembly. Obviously, such models must capture ordered and heterogenous hierarchies of SGML/XML documents as well as the flexible schemas called DTD's

(Document Type Definitions). We further believe that such models must provide patterns and contextual conditions as well as schema transformation. Patterns and contextual conditions are well recognized in the document processing community, while schema transformation has been quite common in the database community. However, to the best of our knowledge, no data models have combined all three of patterns, contextual conditions, and schema transformation simultaneously.

Patterns and contextual conditions help to identify interesting nodes of SGML/XML documents. In our terminology, a *pattern* is a condition on (immediate or non-immediate) subordinate nodes. For example, given PODDP'98 papers, we might want to retrieve sections containing the word "preliminaries" in their titles. Here a pattern is *containing sections whose titles contain "preliminaries"*. In our terminology, a *contextual condition* is a condition on non-subordinates such as (immediate or non-immediate) superiors, siblings, and subordinates of siblings, etc. For example, assume that we are interested only in those sections of document database papers. Then, we want to introduce a contextual condition: *the section must be directly or indirectly subordinate to a paper node such that its title node contains the word "document"*. Many SGML/XML transformation engines [1] support such patterns and contextual conditions. Academic papers on document retrieval via patterns and contextual conditions are surveyed by Baeza-Yates and Navarro [3].

On the other hand, many data models provides schema transformation. For example, the projection operator of the RDB not only creates an instance (a set of tuples) but also a schema, which is a schema with fewer attributes. The created instance is guaranteed to conform to the created schema, although the schema is constructed only from the input schema (without considering the instance). Schema transformation is highly important for at least two reasons. First, a programmer can combine operators repeatedly, since he or she knows the intermediate schema created by each operator. He or she only has to make sure that the next operator is applicable to this intermediate schema. Second, queries designed for a database schema are applicable to any database instance of this schema. The instance returned by the query conforms to the schema created from the database schema.

We believe that schema transformation combined with patterns and contextual conditions will become extremely important for document transformation and assembly. For example, if our operator renames top-level *segments* as *chapters*, the created schema should allow *chapters* as top-level components only and allow *segments* as non-top-level components only. (Here "top-level *segments*" is a contextual condition.) If our operator renames lowest-level *segments* as *topics*, the created schema should allow *topics* as the lowest-level components only. (Here "lowest-level *segments*" is a pattern.) Such schemas allow documents to be transformed and assembled repeatedly. To the best of our knowledge, none of the existing data models provides such schema transformation.

[1] See `http://www.sil.org/sgml/publicSW.html#conversion`.

On the basis of the forest-regular language theory [11, 13] (a branch of the tree automaton theory [7]), we present a new data model that combines patterns, contextual conditions, and schema transformation. A forest is an ordered sequence of trees. A schema is a forest automaton and an instance is a finite set of forests. A pattern is captured by a forest automaton; a contextual condition is captured by a pointed forest representation (a variation of pointed tree representations [12]). Given a query based on a pattern and contextual condition, we can construct a reasonably small schema. In many cases, the constructed schema is *minimally sufficient*; any forest permitted by this schema may be generated from some input instance.

The mathematically clean properties of forest-regular languages help to develop an equivalent algebra and rule-based language. First, the class of forest-regular languages is closed under boolean operations. Along with the fact that an instance is a set of forests rather than a single forest, this closure property provides boolean operators readily. Second, this class is also closed under concatenation and root removal. This closure property provides forest composition and decomposition operators. (Observe that we would lose this property if we used trees rather than forests.) Furthermore, our algebra can easily mimic the relational algebra and any query is in PTIME, although we defer the proof to the full paper for space limitation.

The rest of this paper is organized as follows. Section 2 discusses related work on document database systems. Section 3 introduces forests and forest automatons, and then defines schemas and instances. Sections 4 presents an algebraic language called the forest algebra, and Section 5 demonstrates two examples of document transformation. Section 6 presents a rule-based language called forestlog.

2 Related Work

There have been a number of data models for structured documents, and they are surveyed by Baeza-Yates and Navarro [3]. Most of these models concentrate on retrieval rather than transformation and assembly. Furthermore, little attention has been paid to schema transformation combined with patterns and contextual conditions.

Regarding those works which provide schema transformation, there have been two approaches. One approach uses complex value models [2] or object-oriented models [14]. The other approach [6, 8, 9] uses grammars and parse trees.

Complex value models extend the relational data model by allowing sets and nesting of sets and tuples. Other constructors such as bags and lists are often introduced. Object-oriented models further extend complex value models by introducing OID's, class hierarchies, methods, etc. Both types of models provide algebras, calculus, and rule-based languages, which are equally expressive. They are elegant extensions of the relational algebra, the relational calculus, and datalog, respectively.

Several attempts have been made to capture hierarchies of structured documents with complex value models or object-oriented models. Probably, the most notable example is by Christophides *et al.* [5]. They use O_2 as a basis and further introduce ordered tuples and marked unions so as to capture hierarchies of SGML/XML documents. However, as a result of combining these mechanisms, the representation of a hierarchy becomes complex. For example, the sibling relationship is hard to utilize, as marked unions and OID's intrude. Furthermore, modifications to a DTD, even when the new DTD permits all documents permitted by the current DTD, lead to cumbersome update of database instances. This model introduces contextual conditions on ancestors, but schema transformation does not take full advantage of them, thus providing loose schemas.

An entirely different approach provides grammatical models ([1, 6, 8, 9] among others). A schema in a grammatical model is an extended context-free grammar. An instance of this schema is a derivation tree. This approach naturally captures SGML/XML documents. However, none of the existing models provides schema transformation combined with patterns and contextual conditions. Gonnet and Tompa [8] provide powerful operators, but the result of a query does not have an associated schema. Further, no declarative query languages are provided. Abiteboul, Cluet, and Milo [1] provide schema transformation, but do not provide schema transformation combined with patterns and contextual conditions. Gyssens *et al.* [9] provide an equivalent algebra and calculus, but both are complex and operators are too primitive. Neither their algebra nor calculus are natural extensions of the RDB. Weak patterns are provided, but contextual conditions are not. Colby *et al.* [6] provide a powerful algebra, but does not provide declarative languages. This algebra is not a natural extension of the RDB. Patterns are rather powerful, but contextual conditions are weak. Schema transformation and patterns are combined , but the created schemas tend to be loose (i.e., they allow unnecessary documents).

3 Schemas and Instances

In preparation, we define forests and forest automatons. Let Σ be a finite set of symbols and let X be a finite set of variables. We assume that Σ and X are disjoint and they do not contain $\langle$ or $\rangle$.

A *forest* over Σ and X is a string ($\in (\Sigma \cup X \cup \{\langle, \rangle\})^*$) of the forms as below:

- ϵ (the null forest),
- x ($x \in X$),
- $a\langle u \rangle$ ($a \in \Sigma$, and u is a forest), or
- uv (u and v are forests).

Examples of forests are $a\langle x \rangle$, $a\langle\epsilon\rangle\, b\langle b\langle\epsilon\rangle\, x\rangle$, and $a\langle\epsilon\rangle\, b\langle x\, b\langle b\langle\epsilon\rangle\, x\rangle\, c\langle\epsilon\rangle\rangle$. Observe that symbols in Σ are used as labels of non-leaf nodes of forests and that variables in X are used as those of leaf nodes.

The set of forests over Σ and X is denoted by $\mathbf{F}[\Sigma, X]$. A *tree* is a forest of the form $a\langle u \rangle$. We abbreviate $a\langle\epsilon\rangle$ as a. Thus, the second example is abbreviated as $a\, b\langle b\, x\rangle$.

A *deterministic forest automaton* (DFA) M over a finite set Σ and a finite set X (disjoint from Σ) of variables is a 4-tuple $<Q, \iota, \alpha, F>$, where:

- Q is a finite set of states,
- ι is a function from X to Q,
- α is a function from $\Sigma \times Q^*$ to Q such that for every $q \in Q$ and $x \in \Sigma$, $\{q_1 q_2 \dots q_k \mid k \geq 0, \alpha(a, q_1 q_2 \dots q_k) = q\}$ is regular, and
- F is a regular set (called the *final state sequence set*) over Q.

Given a forest in $\mathbf{F}[\Sigma, X]$, we execute M in the bottom-up manner. First, we assign a state to every leaf node that is labeled with a variable. This is done by computing $\iota(x)$, where x is the variable. Then, we repeatedly assign a state to every node whose subordinate nodes have assigned states. This is done by computing $\alpha(a, q_1 q_2 \dots q_k)$, where a is the label of the node and $q_1 q_2 \dots q_k$ is the sequence of states assigned to the subordinate nodes. Consider the top-level nodes in the given forest and the sequence of those states assigned to them. If this state sequence is an element of the final state sequence set, that forest is *accepted* by M.

The *language accepted* by M, denoted $L(M)$, is the set of forests accepted by M. If a set of forests is accepted by some DFA, this set is *forest-regular*. As in the string case, there are non-deterministic forest automatons and forest-regular expressions. Almost all of the clean properties of regular languages apply to forest-regular languages.

Before we define schemas and instances, we have to consider one difference between the automaton theory and database theory. Although our Σ and X are both finite, the database theory uses countably infinite sets. For example, the RDB theory typically uses a countably infinite set **label** for attributes. Specific RDB schemas or instances use only finite subsets of **label**, but one may introduce new symbols at any time as **label** is countably infinite. The RDB theory uses another countably infinite set **dom**, which contains constants such as integers or strings. An RDB instance has such constants as attribute values.

We follow the database theory approach and use two countably infinite sets **label** and **dom**. They are disjoint and do not contain $\langle$ or $\rangle$. In the SGML/XML terminology, **label** is the set of names and **dom** is the set of character data.

Accordingly, we extend the definition forests. A forest over **label** and **dom** is a string ($\in (\mathbf{label} \cup \mathbf{dom} \cup \{\langle, \rangle\})^*$) of the forms as below:

- ϵ (the null forest),
- x ($x \in \mathbf{dom}$),
- $a\langle u \rangle$ ($a \in \mathbf{label}$, and u is a forest), or
- uv (u and v are forests).

The sets of forests over **label** and **dom** is denoted by $\mathbf{F}[\mathbf{label}, \mathbf{dom}]$.

In the database theory, a special symbol $t (\notin \mathbf{label} \cup \mathbf{dom})$ is typically used in schemas as place holders for constants. An instance of this schema is a finite set of objects that can be obtained by replacing t with constants in **dom**.

We also use a special symbol t for representing schemas. Symbol t is comparable to `#PCDATA` of SGML/XML. Keyword `#PCDATA` occurs only in DTD's,

and do not occur in documents. Documents contain character data when the corresponding portion of the DTD is `#PCDATA`.

Now, we are ready to define schemas and instances. A *schema* is a DFA M over a finite subset of **label** and a singleton $\{t\}$. Two schemas are *equivalent* if they accept the same language. An *instance* over M is a finite subset I of $\mathbf{F}[\mathbf{label}, \mathbf{dom}]$ such that each forest in I is obtained from some forest in $L(M)$ by replacing t with constants in **dom**; difference occurrences of t need not be replaced with the same constant. Note that an instance is not a forest but rather a set of forests. A *database schema* is a collection of schemas $\{M_1, M_2, \ldots, M_m\}$. A *database instance* over this database schema is a collection $\{I_1, I_2, \ldots, I_m\}$, where I_i is an instance of M_i.

4 Forest Algebra

We recursively define *queries*. Let q_1 and q_2 be queries, and let a be a label in **label**. The value of q_1 is denoted $I(q_1)$ and the schema of q_1 is denoted $M(q_1)$. Value $I(q_1)$ is an instance of $M(q_1)$. Observe that schema DFA's can be effectively constructed for all operators.

Basic Values: A variable denoting an (extensional) instance of some schema M is a query. The value is that instance and the schema is M.

Constant values: For every constant c in **dom**, $\{c\}$ is a query. The schema is a DFA that accepts $\{t\}$.

Basic set operations: $q_1 \cap q_2$, $q_1 \cup q_2$, and $q_1 - q_2$ are queries. Their values are defined in the obvious manner. The schemas of $q_1 \cap q_2$ and $q_1 \cup q_2$ are DFA's that accept $L(M(q_1)) \cap L(M(q_2))$ and $L(M(q_1)) \cup L(M(q_2))$, respectively. The schema of $q_1 - q_2$ is $M(q_1)$.

Forest composition and decomposition operations:

- $\mathbf{concat}(q_1, q_2)$ and $\mathbf{addroot}(a, q_1)$ are queries. These operators come from the definition of terms.
  The values are $\{u_1 u_2 \mid u_1 \in I(q_1), u_2 \in I(q_2)\}$ and $\{a\langle u\rangle \mid u \in I(q_1)\}$, respectively. The schema DFA's accept $\{u_1 u_2 \mid u_1 \in L(M(q_1)), u_2 \in L(M(q_2))\}$ and $\{a\langle u\rangle \mid u \in L(M(q_1))\}$, respectively.
- q_1/q_2, $q_1 \backslash q_2$, and $\mathbf{removeroot}(a, q_1)$ are queries. These operators are destructors corresponding to the above constructors.
  The values are $\{v \mid u_1 = vu_2, u_1 \in I(q_1), u_2 \in I(q_2)\}$, $\{v \mid u_1 = u_2 v, u_1 \in I(q_1), u_2 \in I(q_2)\}$, and $\{u \mid a\langle u\rangle \in I(q_1)\}$, respectively. The schema DFA's accept $\{v \mid u_1 = vu_2, u_1 \in L(M(q_1)), u_2 \in L(M(q_2))\}$, $\{v \mid u_1 = u_2 v, u_1 \in L(M(q_1)), u_2 \in L(M(q_2))\}$, and $\{u \mid a\langle u\rangle \in L(M(q_1))\}$, respectively
- $\mathbf{subtree}(q_1)$ is a query. This operator retrieves subtrees. It can be compared to the powerset operator in complex object models, although our operator does not require exponential time.

The value is the set of all trees v such that v is a subtree of some element in $I(q_1)$. The schema is a DFA that accepts $\{v \mid v \text{ is a subtree of } u, u \in L(M(q_1))\}$.

- **prod**(a, q_1, q_2) is a query. This operator constructs higher forests. It comes from the definition of forest-regular expressions, which is beyond the scope of this paper.
  The value is the set of all forests v for which there exist $u_1 \in I(q_1)$ and $u_2 \in I(q_2)$ such that v is obtained by replacing each occurrence of a as a leaf in u_1 by u_2. Different occurrences of a must be replaced with the same forest [2].
  The schema is a DFA that accepts the set of all forests v for which there exists $u_1 \in L(M(q_1))$ such that v is obtained by replacing each occurrence of a as a leaf in u_1 by some element in $L(M(q_2))$.
- **rewrite**(a, i, v, q_1) and **genrewrite**(a, v', q_1) are queries, where i is a non-negative integer, v is a forest over **label** and **dom** $\cup \{\mu_1, \mu_2, \ldots, \mu_i\}$, and v' is a forest over **label** and **dom** $\cup \{\mu_1, \mu_2\}$. These operators rewrite forests and are variations of tree homomorphisms [7].
  The value of **rewrite**(a, i, v, q_1) is the set of forests in $I(q_1)$ rewritten with a, i, v. Each node in a forest is replaced with v, if the label of the node is a and the number of its subordinates is i. The result of rewriting the j-th subtree of the node is assigned to variable μ_j in v $(1 \leq j \leq i)$. Likewise, the value of **genrewrite**(a, v, q_1) is the set of forests in $I(q_1)$ rewritten with a, v. Each node in a forest is replaced with v if the label of the node is a. The result of rewriting the subordinates is assigned to variable μ_1 and the mirror image of that result is assigned to variable μ_2.
  As the schemas of **rewrite**(a, i, v, q_1) and **genrewrite**(a, v, q_1), we would like to construct DFA's that accept $\{$**rewrite**$(a, i, v, u) \mid u \in L(M(q_1))\}$ $\{$**genrewrite**$(a, v, u) \mid u \in L(M(q_1))\}$, respectively. Unfortunately, this is not always possible, since these sets are not always forest-regular. (Just like $\{p^n q p^n \mid n = 1, 2, \ldots\}$ is not regular.) However, we can construct larger but reasonably small DFA's. (Just like we can construct a string automaton that accepts $\{p^m q p^n \mid m, n = 1, 2, \ldots\}$.) Such construction is given by Gécseg and Steinby [7].

Operators based on patterns and contextual conditions:

- **select**(M, q_1) is a query, where M is a deterministic DFA over a finite subset of **label** and a finite subset of **dom** $\cup \{t, \mu\}$. This operator comes from the RDB and pattern matching.
  The value of **select**(M, q_1) is the set of all u in $I(q_1)$ such that u can be obtained from some forest in $L(M)$ by replacing t and μ with constants. Different occurrences of t need not be replaced with the same constant.

[2] Forest regular expressions actually differ from our **prod** operator in that different occurrences of a need *not* be replaced with the same forest. Our operator is designed so that it can be mimicked by our rule-based language *forestlog*.

Meanwhile, all occurrences of μ must be replaced with the same constant, thus providing equality conditions.
The schema of $\mathbf{select}(M, q_1)$ is a DFA that accepts $L(M(q_1)) \cap \{f(u) \mid u \in L(M)\}$, where f is a projection that replaces μ and constants with t.

– $\mathbf{mark}(a, \mathcal{C}, \mathcal{P}, q_1)$ is a query, where a is a label in **label**, $\mathcal{C}$ is a contextual condition (see below), and $\mathcal{P}$ is a pattern (see below). The role of this operator is to locate nodes that satisfy patterns and contextual conditions and then rename these nodes.
This operator is the most complicated in our algebra, but is also the source of its expressiveness. We introduce this operator only informally. Our previous paper [10] shows a formal definition, the algorithms for pattern matches and contextual condition testing, and the effective construction of an output schema ([10] is restricted to binary tress though).
In preparation, we study pointed forests and pointed forest representations (a variation of pointed trees and pointed tree representations [12]). A *pointed forest* over a finite alphabet Σ and a finite set X of variables is a forest over Σ and X such that one node is special and that its subordinates is the null forest. For example, $p\dot{q}r$ and $p\langle\dot{q}r\rangle$ are pointed forests, where $\dot{q}$ is a special node. (Remember that q is an abbreviation of $a\langle\epsilon\rangle$.) A *pointed base forest* is a pointed forest such that its special node is a top-level node. A pointed forest is uniquely decomposed into a sequence of pointed base forests. For example, $p\langle qxr\langle\dot{s}yt\rangle\rangle u$ is uniquely decomposed into $\dot{s}yt, qx\dot{r}, \dot{p}u$.
A *pointed base forest representation* is a triplet (L_1, a, L_2), where L_1, L_2 are forest-regular languages over Σ and X, and a is a symbol in Σ. A set of pointed base forests $\{u_1 \dot{a} u_2 \mid u_1 \in L_1, u_2 \in L_2\}$ is represented by (L_1, a, L_2).
A *pointed forest representation* is a regular expression over a finite set of pointed base forest representations. The represented language is the set of pointed forests w such that the decomposition of w, which is a sequence of pointed base forests, can be derived from the regular expression and pointed base forest representations. For example, $(\mathbf{F}[\Sigma, X], p, \mathbf{F}[\Sigma, X])^*$ represents the set of pointed forests such that the special node and all its superiors are labeled with p.
A *contextual condition* $\mathcal{C}$ is a pointed forest representation over a finite subset of **label** and a finite subset of $\mathbf{dom} \cup \{t\}$. The *envelope* of a node in a forest is the result of removing the subordinates of that node and making the node special. A node in a forest *satisfies* a contextual condition if the envelope of that node is obtained from some element of $L(\mathcal{C})$ by replacing occurrences of t with constants in **dom**.
A *pattern* $\mathcal{P}$ is a pair of (1) a DFA M over a finite subset of **label** and a finite subset of $\mathbf{dom} \cup \{t\}$, and (2) a strongly unambiguous regular expression e that represents the final state sequence of M. (For any string, there is at most one way to generate this string from a strongly unambiguous regular expression. For example, p^* is strongly unambiguous, but $(p^*)^*$ is not.) Some subexpressions of e, including e itself, have associated labels in **label**. A node *matches a pattern* if the subordinate forest of that node is obtained from some forest in $L(M)$ by replacing t with constants in **dom**.

Now, let us define the value of $\mathbf{mark}(a, \mathcal{C}, \mathcal{P}, q_1)$. It is the set of forests that are derived from some forest in $I(q_1)$ as follows: for every node that satisfies $\mathcal{C}$ and matches $\mathcal{P}$, we relabel that node with a and introduce nodes corresponding to labeled subexpressions of strongly unambiguous regular expression e. Each of these nodes has the label associated with the corresponding subexpression.

5 Transformation Examples

This section shows two examples of document transformation. Observe that our queries provide not only database instances but also schemas. The first example demonstrates that our algebra has realistic applications. The second example demonstrates that our schema transformation takes full advantage of patterns and contextual conditions.

5.1 Manipulation of Dictionary Entries

This example is inspired by the OED shortening project [4]. We would like to extract interesting entries of a dictionary. Every entry is represented by an SGML document that conforms to a DTD as below:

```
<!ELEMENT ENTRY       (HEADLINE,SENSE*)>
<!ELEMENT HEADLINE    (#PCDATA)>
<!ELEMENT SENSE       (DEFINITION,QUOTE*)>
<!ELEMENT DEFINITION  (#PCDATA)>
<!ELEMENT QUOTE       (AUTHOR?,TEXT)>
<!ELEMENT AUTHOR      (#PCDATA)>
<!ELEMENT TEXT        (#PCDATA)>
```

We consider each entry as a forest and the dictionary as an instance. We first construct a schema DFA from the above DTD, but we do not present it for space limitation.

We would like to retrieve all entries that contain quotations from Shakespeare. This is done by a query $\mathbf{select}(M, z)$, where z is a variable representing the original dictionary. Pattern M is a deterministic forest automaton $<\{r_0, r_1, r_2\}, \kappa, \beta, L((r_0|r_1|r_2)^* r_2 (r_0|r_1|r_2)^*)\}>$ over Σ and $\{t, \text{“Shakespeare”}\}$ (“Shakespeare” $\in$ **dom**), where

$$\Sigma = \{\texttt{ENTRY}, \texttt{HEADLINE}, \texttt{SENSE}, \texttt{DEFINITION}, \texttt{QUOTE}, \texttt{AUTHOR}, \texttt{TEXT}\},$$

$$\kappa(x) = \begin{cases} r_0 & (x = t) \\ r_1 & (x = \text{“Shakespeare”}), \text{ and} \end{cases}$$

$$\beta(a, u) = \begin{cases} r_2 & (a = \texttt{AUTHOR}, u = r_1) \\ r_2 & (u \in L((r_0|r_1|r_2)^* r_2 (r_0|r_1|r_2)^*)) \\ r_0 & (\text{otherwise}) \ . \end{cases}$$

Other than a shortened dictionary (a set of forests), this query constructs a schema DFA for shortened dictionaries. A minimum DTD constructed from this DFA is as below:

```
<!ELEMENT ENTRY       (HEADLINE,SENSE+)>
<!ELEMENT HEADLINE    (#PCDATA)>
<!ELEMENT SENSE       (DEFINITION,QUOTE*)>
<!ELEMENT DEFINITION  (#PCDATA)>
<!ELEMENT QUOTE       (AUTHOR,TEXT)>
<!ELEMENT AUTHOR      (#PCDATA)>
<!ELEMENT TEXT        (#PCDATA)>
```

5.2 Renaming Top-Level and Bottom-Level Segments

Consider documents containing segments. Each segment contains a title and some paragraphs, and also contains segments recursively. We want to rename top-level segments as sections and lowest-level segments as topics. As in the previous example, we would like to transform not only documents but also schemas.

Our schema is a DFA $<Q, \iota, \alpha, F>$ over Σ and $\{t\}$, where:

$$
\begin{aligned}
\Sigma &= \{\mathtt{Doc}, \mathtt{Seg}, \mathtt{Ttl}, \mathtt{Par}\}, \\
Q &= \{q_{\mathtt{doc}}, q_{\mathtt{seg}}, q_{\mathtt{ttl}}, q_{\mathtt{par}}, q_t, q_{\mathtt{deadend}}\}, \\
\iota(t) &= q_t, \\
\alpha(a,u) &= \begin{cases} q_{\mathtt{doc}} & (a = \mathtt{Doc}, u \in L(q^*_{\mathtt{seg}})) \\ q_{\mathtt{seg}} & (a = \mathtt{Seg}, u \in L(q_{\mathtt{ttl}}\, q^*_{\mathtt{par}}\, q^*_{\mathtt{seg}})) \\ q_{\mathtt{ttl}} & (a = \mathtt{Ttl}, u = q_t) \\ q_{\mathtt{par}} & (a = \mathtt{Par}, u = q_t) \\ q_{\mathtt{deadend}} & (\text{otherwise}), \text{ and} \end{cases} \\
F &= \{q_{\mathtt{doc}}\}\ .
\end{aligned}
$$

and an equivalent DTD is as below:

```
<!ELEMENT Doc  (Seg*)>
<!ELEMENT Seg  (Ttl, Par*, Seg*)>
<!ELEMENT Ttl  (#PCDATA)>
<!ELEMENT Par  (#PCDATA)>
```

First, we rename top-level segments as sections by the **mark** operator. Our contextual condition is a pointed forest representation $(\mathbf{F}[\Sigma, \{t\}], \mathtt{Doc}, \mathbf{F}[\Sigma, \{t\}])$ $(\mathbf{F}[\Sigma, \{t\}], \mathtt{Seg}, \mathbf{F}[\Sigma, \{t\}])$. Our pattern is a minimum DFA that accepts $\mathbf{F}[\Sigma, \{t\}]$ and a strongly unambiguous regular expression representing the final state sequence of this DFA.

Second, we rename lowest-level segments as topics by the **mark** operator again. Our contextual condition is a pointed forest representation $(\mathbf{F}[\Sigma, \{t\}], \mathtt{Doc}, \mathbf{F}[\Sigma, \{t\}])(\mathbf{F}[\Sigma, \{t\}], \mathtt{Seg}, \mathbf{F}[\Sigma, \{t\}])^*$. Our pattern is a minimum

DFA that accepts $\mathbf{F}[\Sigma - \{\mathtt{Seg}\}, \{t\}]$ and a strongly unambiguous regular expression representing the final state sequence of this DFA.

Other than a set of transformed documents, these operators create a schema DFA $<Q', \iota', \alpha', F'>$ over Σ' and $\{t\}$, where:

$$
\begin{aligned}
\Sigma' &= \{\mathtt{Doc}, \mathtt{Seg}, \mathtt{Ttl}, \mathtt{Par}, \mathtt{Sec}, \mathtt{Tpc}\},\\
Q' &= \{q_{\mathtt{doc}}, q_{\mathtt{seg}}, q_{\mathtt{ttl}}, q_{\mathtt{par}}, q_{\mathtt{sec}}, q_{\mathtt{tpc}}, q_t, q_{\mathtt{deadend}}\},\\
\iota'(t) &= q_t,\\
\alpha'(a,u) &= \begin{cases}
q_{\mathtt{doc}} & (a = \mathtt{Doc}, u \in L(q^*_{\mathtt{sec}}))\\
q_{\mathtt{sec}} & (a = \mathtt{Sec}, u \in L(q_{\mathtt{ttl}}\, q^*_{\mathtt{par}}\, (q_{\mathtt{seg}}|q_{\mathtt{tpc}})^*))\\
q_{\mathtt{seg}} & (a = \mathtt{Seg}, u \in L(q_{\mathtt{ttl}}\, q^*_{\mathtt{par}}\, (q_{\mathtt{seg}}|q_{\mathtt{tpc}})^*))\\
q_{\mathtt{tpc}} & (a = \mathtt{Tpc}, u \in L(q_{\mathtt{ttl}}\, q^*_{\mathtt{par}}))\\
q_{\mathtt{ttl}} & (a = \mathtt{Ttl}, u = q_t)\\
q_{\mathtt{par}} & (a = \mathtt{Par}, u = q_t)\\
q_{\mathtt{deadend}} & (\text{otherwise}),\ \text{and}
\end{cases}\\
F' &= \{q_{\mathtt{doc}}\}\ .
\end{aligned}
$$

and a DTD equivalent to this schema is as below:

```
<!ELEMENT Doc  (Sec*)>
<!ELEMENT Sec  (Ttl, par*, (Seg | Tpc)*)>
<!ELEMENT Seg  (Ttl, par*, (Seg | Tpc)*)>
<!ELEMENT Tpc  (Ttl, par*)>
<!ELEMENT Ttl  (#PCDATA)>
<!ELEMENT Par  (#PCDATA)>
```

Observe that this schema allows `Sec` nodes only as subordinates of the `Doc` node and that `Tpc` elements are not allowed to have surordinate `Seg` or `Tpc` elements. In other words, this schema is *minimally sufficient*.

6 Forestlog

Although this data model does not have tuples, it is possible to introduce an equivalent rule-based language called *forestlog*. We introduce forestlog only informally. The formal definition and the equivalence proof are left to the full paper.

The key idea is to impose three conditions so as to enable the conversion of a logic program into an algebraic query. Condition 1: each literal in the body of a standard rule has one and only one variable. Condition 2: the variable in a literal occurs only once in that literal. Condition 3: any variable in a standard rule must occur in some positive literal in the body and also occurs in the rule head.

A *variable* is an element of a countably infinite set **var**. An *intentional predicate* is the name of a relation instance computed by rules. An *extentional predicate* is the name of a relation instance stored in the database.

A *literal* is either:

- $S(u)$ or $\neg S(u)$ (positive literal or negative literal), where S is an extentional or intentional predicate and u is a forest over **label** and **var** $\cup$ **dom** in which only one variable occurs and it occurs once and only once, or
- $M(x)$ ("M matches x"), where M is the first parameter of the operator **select** and x is a variable.

A *rule head* is of the form $S(u)$, where S is an intentional predicate and u is either:

- a forest over **label** and **dom** $\cup$ **var**,
- **prod**(a, x, y), where a is a label, and x, y are variables,
- **rewrite**(a, i, v, x), where a, i, v are the first, second, third parameters of the operator **rewrite**, respectively, and x is a variable,
- **genrewrite**(a, v, x), where a and x are the first and second parameters of the operator **genrewrite**, respectively, and x is a variable,
- **mark**$(a, \mathcal{C}, \mathcal{P}, x)$, where a, $\mathcal{C}$, $\mathcal{P}$ are the first, second, third parameters of the operator **mark**, respectively, and x is a variable.

A *rule* is either a *standard rule* or an *ad-hoc rule*. A standard rule is of the form $head \leftarrow A_1, A_2, \ldots, A_m$, where *head* is a rule head and $A_1, A_2, \ldots, A_m$ are literals. Any variable in the rule must occur in the rule head and must occur in some positive literal in the body.

An ad-hoc rule is of the form $S(x) \leftarrow A, R_1(y)$, where (1) S is an intentional predicate, (2) A is either $R_2(xy)$, $R_2(yx)$, or **subtree**(x, y) ("x is a subtree of y"), and (3) R_1, R_2 are intentional or extensional predicates.

A (non-recursive) *program* is a sequence of rules. More than one rule may have the same predicate in the head. Any intentional predicate used in the body of a rule must be defined by the preceding rules. The intentional predicate defined by the last rule is the *target predicate*. The semantic of programs, rules, head, and literals are defined in the obvious manner.

7 Conclusion

We have presented a data model that provides patterns and contextual conditions as well as schema transformation. Patterns and contextual conditions have been heavily used by SGML/XML transformation engines, while schema transformation is common in the database theory. But none of the previous works provides all three of patterns, contextual conditions, and schema transformation. We believe that this data model provides a theoretical foundation of future SGML/XML database systems.

However, there are many remaining issues. First, we do not really know if our operators are powerful enough. There might be some other useful operator that cannot be mimicked by our operators. Second, in order to perform pattern matching and contextual condition checking without scanning the entire document, we probably have to impose some restrictions on patterns and contextual conditions. Such restrictions help to provide index files for examining patterns and contextual conditions.

Acknowledgment I deeply appreciate Prof. Dirk Van Gucht for his extensive discussion and guidance during my stay in Indiana University. Prof. Ethan Munson and Mr. Paul Prescod gave me very helpful comments on an earlier version of this paper.

References

1. Abiteboul, S., Cluet, S., Milo, T.: Querying and updating the file. VLDB '93 **19** (1993) 73–84
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
3. Baeza-Yates, R., Navarro, G.: Integrating contents and structure in text retrieval. SIGMOD Record **25:1** (1996) 67–79
4. Blake, G., Bray, T., Tompa, F.: Shortening the OED: Experience with a grammar-defined database. ACM TOIS **10:3** (1992) 213–232
5. Christophides, V., Abiteboul, S., Cluet, S., Scholl, M.: From structured documents to novel query facilities. SIGMOD Record **23:2** (1994) 313–324
6. Colby, L., Van Gucht, D., Saxton, L.: Concepts for modeling and querying list-structured data. Information Processing & Management **30:5** (1994) 687–709
7. Gécseg, F., Steinby, M.: Tree Automata. Akadémiai Kiadó (1984)
8. Gonnet, G., Tompa, F.: Mind your grammar: a new approach to modeling text. VLDB '87 **13** (1987) 339–346
9. Gyssens, M., Paredaens, J., Van Gucht, D.: A grammar-based approach towards unifying hierarchical data models. SIAM Journal on Computing **23:6** (1994) 1093–1137
10. Murata, M.: Transformation of documents and schemas by patterns and contextual conditions. Lecture Notes in Computer Science **1293** (1997) 153–169
11. Pair, C., Quere, A.: Définition et etude des bilangages réguliers. Information and Control **13:6** (1968) 565–593
12. Podelski, A.: A monoid approach to tree automata. In *Tree Automata and Languages* North-Holland (1992) 41–56
13. Takahashi, M.: Generalizations of regular sets and their application to a study of context-free languages. Information and Control **27** (1975) 1–36
14. Zdonik, S., Maier, D.: Readings in Object-Oriented Database Systems. Morgan Kaufmann (1990)