December 11, 2000
RT0389
Computer Science 10 pages

Research Report

Extended Path Expressions for XML

MURATA Makoto

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Extended Path Expressions for XML

Makoto Murata
IBM Tokyo Research Lab/IUJ Research Institute
1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan
TEL: +81-46-215-4678 FAX: +81-46-273-7413

mmurata@trl.ibm.co.jp

December 5, 2000

Abstract

Query languages for XML often use path expressions to locate elements in XML documents. Path expressions are regular expressions such that underlying alphabets represent conditions on nodes. Path expressions represent conditions on paths from the root, but do not represent conditions on siblings, siblings of superiors, and descendants of such siblings. In order to capture such conditions, we propose to extend underlying alphabets. Each symbol in an extended alphabet is a triplet $<e_1, a, e_2>$, where a is a condition on nodes, and e_1 (e_2) is a condition on elder (resp. younger) siblings and their descendants; e_1 and e_2 are represented by hedge regular expressions, which are as expressive as hedge automata (hedges are ordered sequences of trees). Such an extended path expression can be evaluated for every element by traversing the XML document three times. Furthermore, given an input schema and a query operation controlled by an extended path expression, it is possible to construct an output schema. This is done by identifying where in the input schema the given pointed hedge representation is satisfied.

1 Introduction

XML [3] has been widely recognized as one of the most important formats on the WWW. XML documents are ordered trees containing text, and thus have structures more flexible than relations of relational databases.

Query languages for XML have been actively studied [1, 9]. Typically, operations of such query languages can be controlled by path expressions. A path expression is a regular expression such that underlying alphabets represent conditions on nodes. For example, by specifying a path expression ($section^*, figure$), we can extract figures in sections, figures in sections in sections, figures in sections in sections in sections, and so forth, where $section$ and $figure$ are conditions on nodes. Based on well-established theories on regular languages, a number of useful techniques (e.g., optimization [2, 5, 10, 14]) for path expressions have been developed.

However, when applied to XML, path expressions do not take advantage of orderedness of XML documents. For example, path expressions cannot locate those <section> elements which have subordinate <figure> elements immediately followed by <table> elements.

On the other hand, industrial specifications such as XPath [6] have been developed. Such specifications address orderedness of XML documents. In fact, XPath can capture the above example. However, these specifications are not driven by any formal models, but rather designed in an ad-hoc manner. Lack of formal models prevents generalization of useful techniques originally developed for path expressions.

As a formal framework for addressing orderedness, this paper shows a natural extension of path expressions. First, we introduce hedge regular expressions, which generate hedges (ordered sequences of ordered trees). Hedge regular expressions are equally expressive as hedge automata (variations of tree automata for hedges). Then, we introduce pointed hedge representations. They are regular expressions such that each "symbol" is a triplet $<e_1, a, e_2>$, where e_1, e_2 are hedge regular expressions and a is a condition on nodes. Intuitively, e_1 represent conditions on elder siblings and their descendants, while e_2 represent conditions on younger siblings and their descendants. As a special case, if every hedge regular expression in a pointed hedge representation generates all hedges, this pointed hedge representation is a path expression.

Given a hedge and a pointed hedge representation, we can determine which node in the hedge matches the pointed hedge representation. For every node, (1) we determine which of the hedge regular expressions in the pointed hedge representation the node matches, (2) we then determine which of the triplets the node matches, (3) and we finally evaluate the pointed hedge representation for every node. The computation time is linear to the number of nodes in hedges.

Another goal of this work is schema transformation. Recall that query operations of relational databases construct not only relations but also schemas. For example, given input schemas (A, B) and (B, C), the join operation creates an output schema (A, B, C). Such output schemas allow further processing of output relations.

It would be desirable for query languages for XML to provide such schema transformations. That is, we would like to construct output schemas from input schemas and query operations (e.g., extract, delete), which utilize pointed hedge representations. To facilitate such schema transformation, we construct a match-identifying hedge automaton from a pointed hedge representation. The computation of this automaton assigns marked states to those nodes which match the pointed hedge representation. Schema transformation is effected by first creating a intersection hedge automaton which simulates this match-identifying hedge automaton and the input schema, and then transforming the intersection hedge automaton as appropriate to the query operation.

The rest of this paper is organized as follows. In Section 2, we consider related works. In preparation, we introduce hedges and hedge automata in Section 3, and then introduce hedge regular expressions in Section 4. In Section 5, we introduce pointed hedges and pointed hedge representations. In Section 6, we study how to locate nodes in hedges by evaluating pointed hedge representations. In Section 7, we construct match-identifying hedge automata from pointed hedge representations, and then construct output schemas. In Section 8, we conclude and consider future works.

2 Related Works

Extensions of path expressions for capturing conditions on siblings have been studied by several researchers [21, 18, 7, 17]. Some recent papers [21, 15, 12] consider schema transformation for XML.

Among these works, [21] is the closest to ours. Its patterns use regular expressions to navigate both vertically and horizontally (i.e., ancestors and siblings). Moreover, it provides schema transformation, which they call "DTD inference". Their input schemas, which they call regular loto (labelled ordered tree object) type definitions, represent hedge *local* languages, while their output schemas represent hedge *context-free* languages. However, in our framework, both input and output schemas represent hedge regular languages rather than hedge local languages. Since all XML schema languages (except XML DTDs) use hedge regular languages rather than hedge local languages [13], we would argue that our work is more applicable to such languages.

Neven [18] introduced pattern languages F$OREG$ and F$OREG^*$. Patterns in these languages have both horizontal path expressions and vertical path expressions. Furthermore, he has established some equivalence between his languages and monadic second-order logic.

In our framework, expressiveness of pointed hedge representations are equivalent to hedge automata. Since monadic second-order logic and tree automata are strongly related, we conjecture that expressiveness of F$OREG^*$ and pointed hedge representations are comparable. Schema transformation is not provided.

Although [18] and [21] allow variables in patterns, our framework does not allow variables at present. As a result, a pointed hedge representation cannot locate tuples of elements. Introduction of such variables are discussed in Section 8.

Given an input DTD and transformation program, Milo et al [15] check whether every result of transformation conforms to a specified output DTD. DTDs are represented by tree automata, and transformations are represented by k-pebble transducers. Path expressions are not used.

Catapillar expressions [4] capture conditions on ancestor nodes, sibling nodes, etc. Expressiveness of catapillar expressions is compared with that of regular tree languages. Schema transformation is not provided.

XDuce [12] is a programming language for handling XML documents. Types in XDuce are regular expressions of types. Operations in XDuce perform regular expression pattern matching. Furthermore, XDuce provides type inference by using tree automata. However, conditions on non-subordinate nodes such as ancestor nodes cannot be captured in XDuce.

3 Hedges and Hedge Automata

In this section, we introduce hedges (ordered sequences of ordered trees) and hedge automata.

Let Σ be an alphabet, and let X be a finite set of variables. We assume that they are disjoint and do not contain either $\langle$ or $\rangle$. A *hedge over* Σ and X is recursively defined below:

- ϵ (the empty hedge),
- x ($x \in X$),
- $a\langle u\rangle$ ($a \in \Sigma$, u is a hedge),
- uv (u and v are hedges).

For example, $a\langle\epsilon\rangle$, $a\langle x\rangle$, $a\langle\epsilon\rangle\, b\langle b\langle\epsilon\rangle\, x\rangle$ are hedges. Note that symbols in Σ are used as labels of non-leaf nodes, while variables in X are used as labels of leaf nodes. Hereafter, we abbreviate $a\langle\epsilon\rangle$ as a. Thus, the third example may be abbreviated as $a\, b\langle b\, x\rangle$.

A *deterministic hedge automaton* M is a 4-tuple $<Q, \iota, \alpha, F>$ such that

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

Given a hedge, we execute a deterministic hedge automaton M in the bottom-up manner. First, we assign a state to every leaf node that is labelled with a variable. This is done by computing $\iota(x)$, where x is the variable. Then, we repeatedly assign a state to each of those nodes such that their subordinate nodes already have states assigned. This is done by computing $\alpha(a, q_1 q_2 \ldots q_k)$, where a is the label of the node and $q_1 q_2 \ldots q_k$ is the sequence of states assigned to the subordinate nodes. Consider the top-level nodes in the given hedge and the sequence of the states assigned to them. If this state sequence is contained by F, the deterministic hedge automaton *accepts* that hedge. For example, $a\langle\epsilon\rangle x$ is accepted if F contains $\alpha(a, \epsilon)$ followed by $\iota(x)$. The *language accepted* by M, denoted $L(M)$, is the set of hedges accepted by M.

Non-deterministic hedge automata are similarly defined. The only difference is that the range of ι and α is the power set of Q. A non-deterministic hedge automaton accepts a hedge if at least one of the possible computations yield a state sequence in F.

4 Hedge Regular Expressions

In this section, we introduce hedge regular expressions, which are as expressive as hedge automata.

Although there are many works [11, 8] on binary tree regular expressions, hedge regular expressions have not been studied in the literature. To the best of our knowledge, the work closest to ours is [20]. Their expressions capture the class of hedge local languages, which is a proper subclass of hedge regular languages. Since any hedge regular language can be obtained by applying some projection to some hedge local language, a pair of an expression and projection provides a hedge regular "expression". Our work differs in not using projections. In other words, our hedge regular expressions directly capture hedge regular languages.

Recall that regular expressions for strings have the concatenation and the closure (*) operator. To introduce hedge regular expressions, we have to provide two pairs of these operators. The first pair creates new hedges by aligning hedges in the horizontal direction. Meanwhile, the second pair creates new hedges by embedding hedges in hedges.

Although it is easy to align hedges in the horizontal direction, it is not straightforward to embed hedges in hedges. Where in a hedge do we embed other hedges? As a target for such embedding, we introduce substitution symbols.

Let Z be a set of substitution symbols. We assume that Z and $\Sigma \cup X$ are disjoint. A hedge over Σ and X with substitution symbols in Z are defined below:

- ϵ,
- x ($x \in X$),
- $a\langle z\rangle$ ($a \in \Sigma, z \in Z$),
- $a\langle u\rangle$ ($a \in \Sigma, u$ is a hedge with substitution symbols)
- u_1u_2 (u_1, u_2 are hedges with substitution symbols)

Let U be a set of hedges with substitution symbols, v be a hedge with substitution symbols, and s be a substitution symbol. Hedges in U are embedded in v by replacing each occurrence of z in v by hedges in U. Different occurrences of z may be replaced by different hedges. The set of hedges obtained by embedding U in v at z is denoted by $U \circ_z v$. When V is a set of hedges with substitution symbols, $U \circ_z V$ is defined as $\bigcup_{v \in V} U \circ_z v$.

Now, we are ready to introduce hedge regular expressions. A hedge regular expression over Σ, X, and Z is defined below:

- ϵ,
- x ($x \in X$),
- $a\langle e\rangle$ ($a \in \Sigma, e$ is a hedge regular expression),
- e_1e_2 (e_1, e_2 are hedge regular expressions),
- $e_1|e_2$ (e_1, e_2 are hedge regular expressions),
- e^* (e is a hedge regular expression),
- $a\langle z\rangle$ ($a \in \Sigma, z \in Z$),
- $e_1 \circ_z e_2$ (e_1, e_2 are hedge regular expressions, and $z \in Z$),
- e^z (e is a hedge regular expression, and $z \in Z$) ,

A hedge regular expression e represents a set $L(e)$ of hedges with substitution symbols. $L(e)$ is recursively defined below:

$$\begin{aligned}
L(\epsilon) &= \{\epsilon\},\\
L(x) &= \{x\},\\
L(a\langle e\rangle) &= \{a\langle u\rangle \mid u \in L(e)\},\\
L(e_1e_2) &= \{u_1 \text{ followed by } u_2 \mid \\
&\qquad u_1 \in L(e_1), u_2 \in L(e_2)\},\\
L(e_1|e_2) &= L(e_1) \cup L(e_2),\\
L(e^*) &= \{\epsilon\} \cup L(e) \cup L(ee) \cup L(eee) \cup \ldots,\\
L(a\langle z\rangle) &= \{a\langle z\rangle\},\\
L(e_1 \circ_z e_2) &= L(e_1) \circ_z L(e_2)\\
L(e^z) &= L(e^{1,z}) \cup L(e^{2,z}) \cup L(e^{3,z}) \cup \ldots
\end{aligned}$$

$$\begin{aligned}
L(e^{1,z}) &= L(e),\\
L(e^{2,z}) &= L(e^{1,z}) \circ_z L(e) \cup L(e^{1,z}),\\
L(e^{3,z}) &= L(e^{2,z}) \circ_z L(e) \cup L(e^{2,z}) \ldots
\end{aligned}$$

For example, consider a hedge regular expression $a\langle z\rangle^{*z}$. Obviously, $L(a\langle z\rangle^*)$ is

$\{\epsilon, a\langle z\rangle, a\langle z\rangle a\langle z\rangle, a\langle z\rangle a\langle z\rangle a\langle z\rangle, \ldots\}$. To compute $L(a\langle z\rangle^{*z})$, we have to compute $L(a\langle z\rangle^{*\,1,z})$, $L(a\langle z\rangle^{*\,2,z})$, $L(a\langle z\rangle^{*\,3,z})$, and so forth.

For every positive integer i, $L(a\langle z\rangle^{*\,i,z})$ contains all hedges such that (1) their height is equal to or less than i, (2) every symbol is a, and (3) every substitution symbol is z. Therefore, $L(a\langle z\rangle^{*z})$ contains all hedges such that (1) every symbol is a, and (2) every substitution symbol is z.

Given a hedge regular expression e, we can always construct a hedge automaton over $\Sigma \cup Z$ that accepts $L(e)$. Likewise, given a hedge automaton M, we can always introduce some substitution symbols and construct a hedge regular expression that accepts $L(M)$. The proof is outside the scope of this paper, but is similar to the proof of equivalence of binary tree expressions and binary tree automata.

5 Pointed Hedge Representations

In this section, we introduce pointed hedge representations, which naturally extend path expressions. Pointed binary tree representations were originally introduced by [22, 19] and their applications to structured documents were studied in [16]. But our pointed hedge representations handle hedges rather than binary trees, and they use hedge regular expressions rather than hedge (or tree) automata.

5.1 Pointed Hedges

In preparation, we introduce some definitions. A *pointed hedge* over alphabet Σ and X is a hedge with one substitution symbol η such that η occurs once and only once. For example, $a\langle x\rangle b\langle \eta\rangle$ and $a\langle x\rangle b\langle c\langle \eta\rangle y\rangle$ are pointed hedges (see Figure 1).

The product of pointed hedges u and v, denoted by $u \oplus v$, is the result of replacing η in v by u. In other words, it is the only element of $\{u\} \circ_\eta v$. For example, the product of $a\langle x\rangle b\langle \eta\rangle$ and $a\langle x\rangle b\langle c\langle \eta\rangle y\rangle$ is $a\langle x\rangle b\langle c\langle a\langle x\rangle b\langle \eta\rangle\rangle y\rangle$.

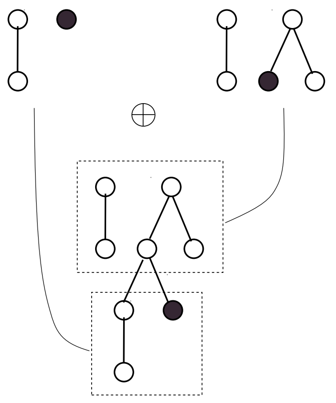

Figure 1: Pointed hedges and their product (The left-top example is $a\langle x\rangle b\langle \eta\rangle$ and the right-top example is $a\langle x\rangle b\langle c\langle \eta\rangle y\rangle$. Their product is $a\langle x\rangle b\langle c\langle a\langle x\rangle b\langle \eta\rangle\rangle y\rangle)$

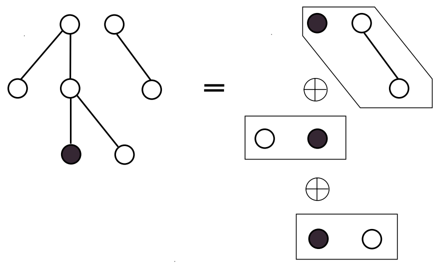

Figure 2: Decomposition of pointed hedges (the right-hand side begins at the bottom and ends at the top).

The associative law holds; that is, $(u \oplus v) \oplus w = u \oplus (v \oplus w)$ for any pointed hedges u, v, w.

A *pointed base hedge* is a pointed hedge of the form $u_1 a\langle \eta\rangle u_2$, where u_1, u_2 are hedges and a is a symbol in Σ. For example, $a\langle x\rangle b\langle \eta\rangle$ is a pointed base hedge, but $a\langle x\rangle b\langle c\langle \eta\rangle y\rangle$ is not. Any pointed hedge can be uniquely decomposed into a sequence of pointed base hedges (see Figure 2). For example, $a\langle x\rangle b\langle c\langle \eta\rangle y\rangle$ can be decomposed into $c\langle \eta\rangle y$ and $a\langle x\rangle b\langle \eta\rangle$.

A *pointed base hedge representation* over alphabet Σ and variable-set X is a triplet $< e_1, a, e_2 >$, where $a \in \Sigma$ and e_1, e_2 are hedge regular expressions. The represented language,

$$e \ni u = \begin{cases} u_k & \in & L(<e_{k1}, a_k, e_{k2}>) \\ \oplus & & \\ \vdots & & \\ \oplus & & \\ u_2 & \in & L(<e_{21}, a_k, e_{22}>) \\ \oplus & & \\ u_1 & \in & L(<e_{11}, a_k, e_{12}>) \end{cases}$$

Figure 3: Matching of pointed hedge representations and pointed hedges

denoted by $L(<e_1, a, e_2>)$, is $\{u_1 a\langle\eta\rangle u_2 \mid u_1 \in L(e_1), u_2 \in L(e_2)\}$.

As an example, consider a pointed base hedge representation $<a\langle z\rangle^{*z}, b, a\langle z\rangle^{*z}>$, where $a\langle z\rangle^{*z}$ is the example hedge regular expression in the previous section. Recall that this hedge regular expression generates all hedges such that every symbol is a and every substitution symbol is z. Therefore, a pointed hedge is generated by $<a\langle z\rangle^{*z}, b, a\langle z\rangle^{*z}>$, when the parent of η is labelled with b, the other nodes are labelled with a, and the substitution symbols are z.

A *pointed hedge representation* over alphabet Σ and variable-set X is a regular expression e over a finite set Δ of pointed base hedge representations. A pointed hedge u matches this pointed hedge representation if e generates a sequence $<e_{11}, a_1, e_{12}>, <e_{21}, a_2, e_{22}>, \ldots <e_{k1}, a_k, e_{k2}>$, u is decomposed into a sequence of pointed base hedges $u_1, u_2, \ldots, u_k$ (i.e., $u = u_1 \oplus u_2 \oplus \ldots \oplus u_k$), and u_i is contained by $L(<e_{i1}, a, e_{i2}>)$ for every i $(1 \leq i \leq k)$. For example,

As an example, consider a pointed hedge representation $<a\langle z\rangle^{*z}, b, a\langle z\rangle^{*z}>^*$. A pointed hedge matches this pointed hedge representation if (1) the parent of η is labelled with b, (2) all its ancestor nodes are labelled with b, (3) all other nodes are labelled with a, and (4) the substitution symbols are z.

Just like path expressions locate nodes, a pointed hedge representation r locates nodes in hedges. For each node in a hedge, we construct a pointed hedge by replacing the subordinates of this node by η. If this pointed hedge mathces r, this node is located by r.

Finally, we would like to point out that expressiveness of pointed hedge representations (to be precise, pointed binary tree representations) have been already studied [22, 19]. Equivalence of monoid recognizability (pointed binary tree representations) and recognizability (binary tree automata) has been established. Hence, we would argue that pointed hedge representations are as powerful as possible in the framework of hedge automata.

6 Evaluation of Pointed Hedge Representations

We show an algorithm for evaluating pointed hedge representations. Given a hedge, this algorithm locates those nodes which satisfy the pointed hedge representation by traversing the hedge three times.

Recall that a pointed hedge representation is a regular expression over a finite set Δ of pointed base hedge representations. Let Δ be $\{<e_{11}, a_1, e_{12}>, <e_{21}, a_2, e_{22}>, \ldots, <e_{n1}, a_n, e_{n2}>\}$.

For each e_{i1} and e_{i2} $(1 \leq i \leq n)$, we construct deterministic hedge automata M_{i1} and M_{i2}.

Without loss of generality, we can assume that M_{i1}, M_{i2} $(1 \leq i \leq n)$ share the state set Q, the transition function ι, and the transition function α. That is,

$$\begin{array}{rcll} M_{i1} & = & <Q, \iota, \alpha, F_{i1}> & (1 \leq i \leq n), \\ M_{i2} & = & <Q, \iota, \alpha, F_{i2}> & (1 \leq i \leq n) \end{array}$$

If they did not share Q, ι, α, we only have to use the cross product of all state sets as a new state set; that is, we use $Q_{11} \times Q_{12} \times Q_{21} \times Q_{22} \times \ldots Q_{n1} \times Q_{n2}$ as the state set, where Q_{ij} is the state set of M_{ij}. We then reconstruct transition functions and final state sequences for this new state set.

Again, without loss of generality, we can assume that $F_{i1} = F_{j1}$ or $F_{i1} \cap F_{j1} = \emptyset$, and that $F_{i2} = F_{j2}$ or $F_{i2} \cap F_{j2} = \emptyset$ for every i, j $(i \neq j)$. If this assumption does not hold, we only have to construct hedge regular expressions for $F_{i1} \cap \neg F_{j1}$, $F_{i1} \cap F_{j1}$, $\neg F_{i1} \cap F_{j1}$, $F_{i2} \cap \neg F_{j2}$, $F_{i2} \cap F_{j2}$, $\neg F_{i2} \cap F_{j2}$, and rewrite the original pointed hedge representation.

Given a pointed base hedge $u_1 a\langle\eta\rangle u_2$ (u_1 and u_2 are hedges), we would like to know which pointed base hedge representation $<e_{i1}, a_i, e_{i2}>$ the pointed base hedge matches. First, we assign a state to each node in u_1 and u_2 by evaluating the transition functions ι and α. Let $\lceil u_1 \rceil$ be the state sequence assigned to the top-level node sequence of u_1, and let $\lceil u_2 \rceil$ be the state sequence assigned to the top-level node sequence of u_2. Pointed base hedge $u_1 a\langle\eta\rangle u_2$ matches $<e_{i1}, a_i, e_{i2}>$ if and only if $\lceil u_1 \rceil$ is contained in F_{i1}, a is equal to a_i, and $\lceil u_2 \rceil$ is contained in F_{i2}.

A pointed base hedge does not match more than one pointed base hedge representation. If it matches $<e_{i1}, a_i, e_{i2}>$ and $<e_{j1}, a_j, e_{j2}>$, we have a state sequence contained by F_{i1} and F_{j1} and another state sequence contained by F_{i2} and F_{j2}, and $a = a_i = a_j$. Then, by our hypotheses, these two pointed base hedge representations are identical.

We further assume that any pointed base hedge match with some pointed base hedge representation δ in Δ. If this assumption does not hold, we only have to add more pointed base hedge representations to Δ.

We can now construct a classification function θ from $Q^* \times \Sigma \times Q^*$ to Δ. Given state sequence $q_{11}q_{12}\ldots q_{1i}$, symbol a, and state sequence $q_{21}q_{22}\ldots q_{2j}$, this function chooses the pointed base hedge representation $<e_{k1}, a_k, e_{k2}> \in \Delta$ such that $a_k = a$, and $q_{11}q_{12}\ldots q_{1i}$ and $q_{21}q_{22}\ldots q_{2j}$ are contained by F_{k1} and F_{k2}, respectively. The pointed base hedge representation that matches a pointed base hedge $u_1 a\langle\eta\rangle u_2$ is $\theta(\lceil u_1 \rceil, a, \lceil u_2 \rceil)$.

Remember that a pointed hedge representation e generates a regular set over a finite set of pointed base hedge representations. Consider the mirror image of this set, namely $\{w_k \ldots w_2 w_1 \mid \quad w_1 w_2 \ldots w_k \text{ is generated by } e\}$. Since the mirror image of a regular set is regular, we can construct a deterministic automaton $N = <S, \mu, s_0, S_{\text{fin}}>$ that accepts this set, where S is a finite set of states, s_0 ($\in S$) is a start state, and S_{fin} ($\subseteq S$) is a set of final states.

Now, we are ready to introduce an algorithm for locating those nodes which satisfy a pointed hedge representation. First, to each node, we assign a state in Q by evaluating transition functions ι and α. Then, to each node, we assign a pointed base hedge representation in Δ by evaluating the classification function θ; that is, we classify the pointed base hedge comprising the elder siblings (including their subordinates), the node (which is assumed to have η as the subordinate), and the younger siblings (including their subordinates). Finally, to each node, we assign a state in S by applying μ to the assigned pointed base hedge representation and the state of its parent node. If and only if a final state in S_{fin} is assigned to a node, this node is located by the given pointed hedge representation. Obviously, this algorithm takes time linear to the number of nodes.

7 Construction of Match-identifying Hedge Automata

In this section, to facilitate schema transformation, we construct a match-identifying automaton from a pointed hedge representation.

Suppose that query operations (e.g., delete) are controlled by pointed hedge representations. Given an input schema and such a query operation, we would like to construct an output schema. For this purpose, we have to identify where in the input schema the given pointed hedge representation is satisfied.

The match-identifying hedge automaton constructed from the pointed hedge representation accepts any hedge. But the match-identifying hedge automaton assigns a *marked state* to each node in a hedge, if and only if the node satisfies the pointed hedge representation. From the input schema and the match-identifying hedge automaton, we can construct an intersection hedge automaton. This hedge automaton accepts the same language as the input schema does, but further identifies matches by marked states. By modifying this automaton as appropriate to the query operation, we can generate an out schema. In the case of extraction, we only have to use marked states as final state sequences.

Careful readers might wonder whether construction of such match-identifying hedge automata is possible. Hedge automata are bottom-up, but pointed hedge representations capture conditions on non-subordinates. Deterministic hedge automata cannot predict what they will encounter later. To overcome this problem, we use non-deterministic unambiguous hedge automata.

The key idea is to make a non-deterministic automaton N' which simulates N in reverse (Fig 4). That is, (1) if N has a transition labelled δ from a state s_1 to another state s_2, then N' has a transition labelled δ from s_2 to s_1 via δ, where δ is a pointed base hedge representation, (2) the start state of N is a final state of N', and (3) the final states of N are start states of N'. Formally, N' is defined as $<S, \mu', S_{\text{fin}}, \{s_0\}>$, where $\mu'(\delta, s_2) \ni s_1$ if and only if $\mu(\delta, s_1) = s_2$, S_{fin} is the set of start states, and $\{s_0\}$ is the set of final states.

Suppose that N' has successful computations (sequences of states) for a string. If we reverse these computations, we obtain computations of N for the mirror image of this string. Since N is a deterministic automaton, it has only one computation per string. Therefore, N' is unambiguous: it has only one successful computation per string. At each symbol in a string, N' may have multiple choices. But only one of them leads to a final state.

We construct a match-identifying automaton $<Q', \kappa, \beta, F'>$. First, the set of states Q' is defined below:

$$Q' = (Q \times S \times \Sigma) \cup (Q \times \{s_\perp\} \times \{a_\perp\}).$$

Since we intend to use the classification function θ in the definition of β, a state in Q' comprises a symbol in Σ. Use of S and Q in Q' allow simulation of N and M_{i1}, M_{i2}, respectively. $s_\perp$ and $a_\perp$ are additional values for leaf nodes.

A set Q'_{mark} of marked states is defined below:

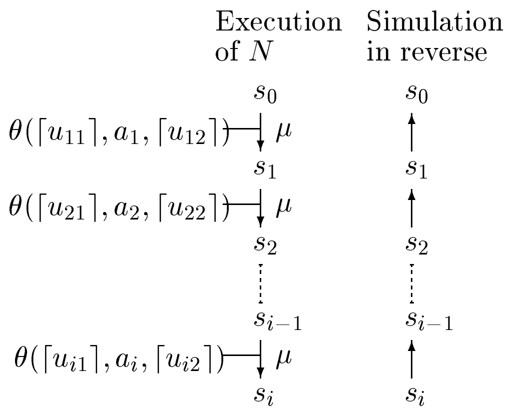
Execution of N
Simulation in reverse
s_0
$\theta(\lceil u_{11}\rceil, a_1, \lceil u_{12}\rceil)$
μ
s_1
$\theta(\lceil u_{21}\rceil, a_2, \lceil u_{22}\rceil)$
μ
s_2
s_{i-1}
$\theta(\lceil u_{i1}\rceil, a_i, \lceil u_{i2}\rceil)$
μ
s_i

Figure 4: Simulation of N in reverse

$$Q'_{\text{mark}} = Q \times S_{\text{fin}} \times \Sigma.$$

Use of S_{fin} implies that N' begins with one of its start states.

Mapping κ from Σ to Q' is defined below:

$$\kappa(x) = <\iota(x), s_\perp, a_\perp> .$$

Next, we define a function β from $\Sigma \times Q'^*$ to the power set of Q'. Given a symbol and a sequence of states in Q', this function returns a set of states in Q'. The first ingredient (elements in Q) simulates α. The second ingredient (elements $(S \cup \{s_\perp\})$) simulates N'. The third ingredient (elements in $(\Sigma \cup \{a_\perp\})$ is the symbol given as input.

$$\begin{aligned}
&\beta(a, <q_1, s_1, a_1><q_2, s_2, a_2> \ldots <q_i, s_i, a_i>) = \\
&\quad \{<\alpha(a, q_1 q_2 \ldots q_i), s, a> \mid \\
&\qquad \text{either } s \in \mu'(\theta(q_1 \ldots q_{j-1}, a_j, q_{j+1} \ldots q_i), s_j) \\
&\qquad \text{or } a_j = a_\perp \text{ for every } j\ (1 \le j \le i)\}
\end{aligned}$$

Next, we show that β satisfies the regularity condition. The inverse image of α, namely $\{q_1 q_2 \ldots q_i \mid \alpha(a, q_1 q_2 \ldots q_i) = q\}$ $(a \in \Sigma, q \in Q\}$, is abbreviated as $\alpha_a^{-1}(q)$. We use the same abbreviation for β.

Let us construct the inverse image of β. We can easily show that $\beta_a^{-1}(<q, s, b>$ $(a, b \in \Sigma, a \neq b, q \in Q, s \in S)$ and $\beta_a^{-1}(<q, s_\perp, a_\perp>$ $)$ $(q \in Q)$ are regular.

$$\begin{aligned}
\beta_a^{-1}(<q, s, b>) &= \emptyset \\
\beta_a^{-1}(<q, s_\perp, a_\perp>) &= \emptyset
\end{aligned}$$

Next, we consider $\beta_a^{-1}(<q,s,a> \quad (q \in Q, s \in S, a \in \Sigma)$. Our previous definition of β imposes conditions on each of the child nodes. Rather, by eliminating those child nodes which do not satisfy conditions, we have the following equation.

$$\beta_a^{-1}(<q,s,a>) = h(\alpha_a^{-1}(q)) - \bigcup_{1 \leq i,j \leq n} h(F_{i1}) X_{ij} h(F_{j2})$$

Here X_{ij} is a set defined below:

$$X_{ij} = \{<q,s',a'>|\ s \notin \mu'(\theta(f_{i1},a',f_{j2}),s'), q \in Q \text{ for every } f_{i1}(\in F_{i1}) \text{ and } f_{j2}(\in F_{j2})\}$$

h is a natural extension of a substitution defined below:

$$h(q) = (\{q\} \times S \times \Sigma) \cup \{<q,s_\perp,a_\perp>\}$$

If a state sequence $<q_1,s_1,a_1> <q_2,s_2,a_2> \ \ldots\ <q_i,s_i,a_i>$ is contained by $h(F_{i1})X_{ij}h(F_{j2})$, then s is not contained by $\mu'(\theta(q_1 \ldots q_{j-1}, a_j, q_{j+1} \ldots q_i), s_j)$.

Since regular sets are closed under substitutions, concatenation, and boolean operations, the inverse image of β is a regular set.

Finally, we construct a final state sequence set F'. The first ingredient simulates F. The second ingredient (elements in $(S \cup \{s_\perp\})$) and third ingredient (elements in $(\Sigma \cup \{a_\perp\})$ are defined so that the application of μ to the result of the classification function θ and the final state s_0 of N' yields the second ingredient (elements in $(S \cup \{s_\perp\})$.

$$F' = \{<q_1,s_1,a_1><q_2,s_2,a_2> \ldots <q_i,s_i,a_i> \mid s_j = \mu(\theta(q_1 \ldots q_{j-1},a_j,q_{j+1} \ldots q_i),s_0) \text{ or } a_j = a_\perp \text{ for every } j\ (1 \leq j \leq i)\}$$

It remains to show that F' is regular. As in the construction of the inverse image of β, we can rewrite F as below:

$$F' = h(Q^*) - \bigcup_{1 \leq i,j \leq n} h(F_{i1}) Y_{ij} h(F_{j2})$$

Here Y_{ij} is a set defined below:

$$Y_{ij} = \{<q,s',a'>|\ \mu(\theta(f_{i1},a',f_{j2}),s_0) \neq s', q \in Q \text{ for every } f_{i1}(\in F_{i1}) \text{ and } f_{j2}(\in F_{j2})\}$$

Again, since regular sets are closed under substitutions, concatenation, and boolean operations, F' is a regular set.

8 Conclusions and Future Works

Pointed hedge representations are natural extensions of path expressions such that conditions on siblings, siblings of ancestors, and even their descendants can be represented. We have presented an algorithm for locating nodes by extended path expressions. To generate output schemas from input schemas and query operations, we have constructed match-identifying automata from extended path expressions.

There are some open issues. First, is it possible to generalize useful techniques (e.g., optimization) developed for path expressions to extended path expressions? Second, pointed hedge representations are probably too powerful to implement. In fact, the construction of match-identifying automata is very complicated. It will become significantly easier if we use usual path expressions only. Some restrictions on pointed hedge representations might make the construction easier while preserving expressiveness enough for users. Third, we would like to introduce variables to pointed hedge representations so that query operations can use the values assigned to such variables. For this purpose, we have to study unambiguity of hedge regular expressions and pointed hedge representations. An ambiguous expression may have more than one ways to match a given hedge, while an unambiguous expression has at most one such way. Variables can be safely introduced to unambiguous expressions.

9 Acknowledgements

I appreciate Kilho Shin who pointed me to hedge regular languages. Dongwon Lee and Murali Mani read an earlier draft and have provided many useful comments.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, 1999.

[2] S. Abiteboul and V. Vianu. Regular path queries with constraints. In *PODS 97*, 1997.

[3] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. W3C Recommendation. http://www.w3.org/TR/REC-xml, Feburary 1998.

[4] A. Brüggemann-Klein and D. Wood. Caterpillars: A context specification technique. *Markup Languages: Theory and Practice*, 2(1):81–106, Winter 2000.

[5] P. Buneman, W. Fan, and S. Weinsten. Path constraints on semistructured and structured data. In *PODS 98*, 1998.

[6] J. Clark and S. DeRose. XML Path Language (XPath) version 1.0 W3C Recommendation. http://www.w3.org/TR/xpath, November 1999.

[7] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion! In *SIGMOD 98*, 1998.

[8] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. "Tree Automata Techniques and Applications", 1997. http://www.grappa.univ-lille3.fr/tata.

[9] M. Fernandez, J. Simeon, and P. W. (editors). XML query languages: Experiences and exemplars. http://www-db.research.bell-labs.com/user/simeon/xquery.html.

[10] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *ICDE 98*, 1998.

[11] F. Gécseg and M. Steinby. *Tree Automata.* Akadémiai Kiadó, Budapest, 1984.

[12] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. In *POPL 01*, 2001.

[13] D. Lee, M. Mani, and M. Murata. Reasoning about XML schema languages using formal language theory, November 2000. Technical Report, IBM Almaden Research Center, RJ#10197, Log#95071, http://www.cobase.cs.ucla.edu/tech-docs/dongwon/ibm-tr-2000.pdf.

[14] A. Mendelzon and P. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6), 1995.

[15] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *PODS 00*, 2000.

[16] M. Murata. Transformation of documents and schemas by patterns and contextual conditions. In *PODP 96*, volume 1293 of *LNCS*. Springer-Verlag Inc., 1997.

[17] F. Neven. Extension of attribute grammars for structured document queries. In *DBPL 99*, 1999.

[18] F. Neven and T. Schwentick. Expressive and ecient pattern languages for tree-structured data. In *PODS 00*, 2000.

[19] M. Nivat and A. Podelski. Another variation on the common subexpression problem. *Discrete Mathematics*, 114:379–401, 1993.

[20] C. Pair and A. Quere. Définition et etude des bilangages réguliers. *Information and Control*, 13(6):565–593, Dec. 1968.

[21] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *PODS 00*, 2000.

[22] A. Podelski. A monoid approach to tree automata. In M. Nivat and A. Podelski, editors, *Tree Automata and Languages*, pages 41–56. North-Holland, 1992.