パスオートマトンによるXML文書への型情報の付与

村田 真[†,††]

XMLデータベース検索言語XQueryは，スキーマに基づいた型情報がXML文書に付与されていることを前提とする．型情報を付与する処理は，XMLスキーマ言語W3C XML Schemaに基づく検証の一部として行われる．しかし，この方法にはW3C XML Schema以外のスキーマ言語（とくにRELAX NG）に利用できないという問題点，検証という重い処理が必要になるという問題点がある．これらの問題点を解決するため，パスオートマトンによってXML文書に型情報を付与することを提案する．最初に，RELAX NGなどで書かれたスキーマからパスオートマトンを構築する．次に，パスオートマトンをXML文書に対して実行することにより，XML文書に対して型情報を付与する．

Assigning Types to XML Documents Using Path Automata

Makoto Murata[†,††]

In XQuery, it is assumed that XML documents are accompanied with type information obtained from schemas. Assigning type information is done as part of W3C XML Schema validation. However, this approach has two problems: (1) other schema languages including RELAX NG are not supported, and (2) W3C XML Schema validation, which is very expensive, becomes mandatory. To overcome these problems, this paper proposes path automata as a mechanism for assigning types to XML documents. First, path automata are constructed from schemas written in schema languages such as RELAX NG. Then, type assignment is performed by executing the constructed path automata against XML documents.

1. はじめに

XQuery[1]は，W3Cで制定中のXMLデータベース検索言語である．XQueryが対象とするXML文書は，XML 1.0[2]に規定されているXML文書とは厳密には異なる．すなわち，XML 1.0の構文解析によって得られる木はスキーマに基づく型情報を含まないが，XQueryの検索対象となる木はスキーマに基づく型情報を含む．型情報は，構文解析を行った後に，スキーマ言語W3C XML Schema[3)~5)]に基づく検証によって得られる（ただし他の方法が排除されているわけではない）．

型情報は，大きく2つに分かれる．1つは，`xsd:int`，`xsd:boolean`のような単純型（simple type）である．XML文書に現れる文字列が，実際にはどんなデータを表しているのかは単純型によって定まる．もう1つは，複合型（complex type）である．複合型は，要素がどんな子要素やテキストを持つかについての制約である．W3C XML Schemaでは，複合型宣言（complex type declaration）によって宣言される．

W3C XML Schemaに基づく検証によって型情報を得るという方法には，2つの問題がある．1つは，W3C XML Schema以外のスキーマ言語，とくにRELAX NG[6)]に適用できないという問題点がある．調査[7)]によれば，2003年にWWW上にあるXML文書のうちW3C XML Schemaを利用しているものは0.09%にすぎない☆．

もう1つの問題点は，検証という重い処理が必須になることである．大手ユーザにおけるXMLデータベースの性能について調査し，文献8)は次のように報告している．

- XMLデータベースの性能が低い要因の1つは構文解析と検証である．
- W3C XML Schemaスキーマの内部形式をキャッシュしない検証（Xerces-Cによるもの）には構文解析の1〜2倍の時間を要する．

† 日本IBM株式会社東京基礎研究所
IBM Tokyo Research Laboratories
†† 国際大学研究所
International University of Japan

☆ スキーマ言語RELAX NGについては，文献7)は調査結果を示していない．なお，RELAX NGを用いた場合には，XML文書を調べてもRELAX NGを用いているかどうか判定できない．

- スキーマをキャッシュすればより高速になるが，スキーマが多い場合には有効ではない．

また，川口[9]はスキーマキャッシングをともなう検証（Xercese-J による）であっても構文解析の 0.5 倍から 3 倍の時間を要することを報告している．

これら 2 つの問題点を解決するため，本論文では，型情報の付与を検証から分離する方式を提案する．型情報の付与は，パスオートマトンによって行う．これは，ルート要素からのパスを走査して要素・属性の型を決めるオートマトンであり，RELAX NG などで書かれたスキーマから構築される．本方式は，RELAX NG や W3C XML Schema を含む多くのスキーマ言語に適用できる．また，検証を行うことなしに，型情報の付与だけを実行することができる．パスオートマトンによって型を一意に確定できない場合も存在するが，部分的な検証と不確定を表す型の利用とによって対処する方法を示す．

以下に，本論文の構成を示す．2 章では，スキーマを正規木文法および単一型木文法として定式化する．3 章では，パスオートマトンをスキーマから構築する方法を示す．4 章では，パスオートマトンを用いて要素に型情報を付与する方法を示す．5 章では，スキーマ言語 RELAX NG に限定して，属性・単純型・混在内容モデルをパスオートマトンによって扱う方法を示す．6 章では，RELAX NG スキーマからパスオートマトンを構築するプログラムの実装について述べる．7 章では，本方法の有効性と限界について論じる．

2. スキーマ

本論文では，スキーマを正規木文法によって表現する．DTD，W3C XML Schema，RELAX NG を含む多くのスキーマ言語が，木オートマトンによって表現可能なことが知られている[10]．

本章では要素だけを取り上げて，特定のスキーマ言語に依存しないように一般的に説明する．

2.1 正規木文法

正規木文法は四つ組 $G = (N, \Sigma, S, P)$ である．ここで，

- N は**非終端記号**の有限集合，
- Σ は**名前**の有限集合，
- S は $\Sigma \times N$ の部分集合，
- P は**生成規則** $x \to r$ の有限集合である（$x \in N$，r は $\Sigma \times N$ の上の正規表現）．複数の生成規則が左辺の非終端記号を共有することは，本論文では許さない☆．

生成規則を一回適用することによって，$a[x]$ の形の式（$a \in \Sigma$，$x \in N$）を，$a[b_0[y_0]b_1[y_1]\ldots b_n[y_n]]$ の形の式（$b_i \in \Sigma$，$y_i \in N$）の形の式に置き換えることができる．生成規則の適用を繰り返すことによって，S に属する式から，Σ の上の木を生成することができる．

非終端記号と名前を分離する理由は，同一の名前を持つ要素に対して，子要素についての制約をいくつか使い分けるためである．DTD ではこの分離を必要としないが，W3C XML Schema や RELAX NG は必要とする．

例 1 簡単なスキーマを表す正規木文法 $G_1 = (N_1, \Sigma_1, S_1, P_1)$ を示す．ここで

$$\begin{aligned} N_1 &= \{\text{Doc, Para}\}, \\ \Sigma_1 &= \{\texttt{doc}, \texttt{para}\}, \\ S_1 &= \{\texttt{doc}[\text{Doc}]\}, \\ P_1 &= \{\text{Doc} \to \texttt{para}[\text{Para1}]^*, \\ &\qquad \text{Para1} \to \epsilon\}. \end{aligned}$$

このスキーマと等価な DTD を次に示す．

```
<!ELEMENT doc       (para*)>
<!ELEMENT para      EMPTY>
```

この DTD に対して妥当な XML 文書として，

```
<doc>
  <para/>
  <para/>
</doc>
```

などがある． □

2.2 解　　釈

正規木文法 G による木の解釈とは，各要素に非終端記号を 1 つ付与したものであって，以下の条件を満たすものである．

- ルート要素のラベルが a_{root} であり，付与された非終端記号を x_{root} とすると，$a_{\text{root}}[x_{\text{root}}]$ は S に属する．
- どの要素 e とその子要素 $e_1, e_2, \ldots, e_n$ についても，以下の条件を満たす生成規則 $x \to r$ が存在する．
  - e に付与された非終端記号が x である．
  - $e_1, e_2, \ldots, e_n$ のラベルが $a_1, a_2, \ldots, a_n$ であり，付与された非終端記号が $x_1, x_2, \ldots, x_n$ であるとすると，$a_1[x_1]a_2[x_2]\ldots a_n[x_n]$ は正規表現 r にマッチする．

ある木に対してスキーマ G に基づく解釈が少なく

☆ この制限は，本質的なものではないが，正規木文法と単一型木文法との比較を簡単にするために導入する．

とも 1 つ存在するとき，この木は G に照らして**妥当**であるという．

例 2 XML 文書

```
<doc>
  <para/>
  <para/>
</doc>
```

の要素 doc に非終端記号 Doc を付与し，2 つの要素 para に非終端記号 Para1 を付与すれば，G_1 に基づく解釈を得ることができる（いいかえれば，この文書は妥当である）．なんとなれば doc[Doc] は S_1 に属し，para[Para1]para[Para1] は非終端記号 Doc を持つ生成規則の右辺 para[Para1]* にマッチし，ϵ は非終端記号 Para1 を持つ生成規則の右辺 ϵ にマッチするからである． □

木の正規木文法による解釈は，型情報を付与したものと見なすことができる．ここでいう型とは，非終端記号である．上の例では，ルート要素に型 Doc がルート要素の型であり，Para1 が 2 つの para 要素の型である．

2.3 単一型木文法

以下の条件を満たす正規木文法 $G = (N, \Sigma, S, P)$ を**単一型木文法**という[10]．

(1) S が $a[x]$ と $b[y]$ を含み，$a = b$ であるなら，$x = y$ である．

(2) 生成規則 $x \to r$ の右辺 r に $a[y]$ と $b[z]$ が出現し，$a = b$ であるなら，$y = z$ である．

例 1 の正規木文法が単一型木文法であることを示す．S_1 は root[Root] しか含まないので条件 (1) は成立する．条件 (2) が成立することは，生成規則の右辺に para[Para1] しか出現しないことから分かる．

W3C XML Schema は，単一型木文法だけをスキーマとして認めている☆,☆☆．一方，RELAX NG は任意の正規木文法を許している．

2.4 単一型木文法と正規木文法の性質

単一型木文法と正規木文法については，以下の性質が知られている[10]．

ブール演算に関する閉包性 単一型木文法で表現できる言語のクラスはブール演算について閉じていないが，正規木文法で表現できる言語のクラスは閉じている．

解釈の一意性 単一型木文法は 1 つの木にたかだか 1 つの解釈を与えるが，正規木文法は複数の解釈を与えることがある．

前者は，正規木文法が XML プログラミング言語（XDuce[11] など）の静的型検査に適していることを意味する．一方，型情報の付与に直接関連するのは後者である．複数の解釈を与える正規木文法の例を次に示す．

例 3 正規木文法 $G_2 = (N_2, \Sigma_2, S_2, P_2)$ を考える．この文法が単一型木文法でないことは，a[OptB] と a[B] に注目すれば容易に分かる．

$$
\begin{aligned}
N_2 &= \{\text{Root, B, OptB, Emp}\},\\
\Sigma_2 &= \{\texttt{root}, \texttt{a}, \texttt{b}\},\\
S_2 &= \{\texttt{root}[\text{Root}]\},\\
P_2 &= \{\text{Root} \to (\texttt{a}[\text{OptB}]\texttt{a}[\text{B}])|(\texttt{a}[\text{B}]\texttt{a}[\text{OptB}]),\\
&\quad\ \text{B} \to \texttt{b}[\text{Emp}],\\
&\quad\ \text{OptB} \to \texttt{b}[\text{Emp}]?,\\
&\quad\ \text{Emp} \to \epsilon\ \}.
\end{aligned}
$$

以下の文書を考える．

```
<root>
  <a><b/></a>
  <a><b/></a>
</root>
```

この文書には 2 つの解釈が存在する．1 つの解釈は，1 番目と 2 番目の <a> にそれぞれ B と OptB を付与する．もう 1 つの解釈は，それぞれに OptB と B とを付与する． □

正規木文法は，解釈の一意性を保証しないので，非終端記号を型とすることがつねに可能なわけではない．一方，単一型木文法は一意性を保証しているので，非終端記号をそのまま型として使用することができる．

正規木文法では，解釈を構築することも容易ではない．第一に，解釈が 1 つしか存在しない文書であっても，文書全体を調べ終わるまでは，先頭に現れる要素の非終端記号を決められないことがある．

例 4 以下の文書を G_2 に照らして検証することを考える．

```
<root>
  <a><b/></a>
  <a></a>
</root>
```

2 番目の <a> 要素が子要素を持たないことが分かってはじめて，1 番目の <a> 要素に付与すべき非終端記号が B であって OptB ではないことが確定する． □

☆ 厳密には，W3C XML Schema のワイルドカード（any）を使用すると，単一型木文法ではないスキーマを書くことができるが，本論文では考察しない．

☆☆ W3C XML Schema は決定的内容モデルという制約を DTD から引き継いだため，単一型木文法であっても表現できないことがある．

第二に，複数の解釈が存在する場合に，すべての解釈を構築することは容易ではない．文書全体を見終わるまで，解釈がいくつ存在するかが確定しない．また，きわめて多くの解釈が成り立つこともある．

正規木文法に基づく検証器は，必ずしも文書の解釈を構築しているわけではない．たとえば，RELAX NG の検証器 Jing は，解釈が 1 つ以上存在するかどうかを調べているだけである．いいかえると，これらの検証器の動作は，非決定的オートマトンの実行に似ている．非決定的オートマトンの実行では，終了状態に到着したかどうかだけを調べており，初期状態から終了状態に至る経過をすべて記録しているわけではない．

3. パスオートマトン

本章では，正規木文法からパスオートマトンを構築する方法を示す．また，パスオートマトン構築および実行についての計算量について論じる．

3.1 パスオートマトンの構築

パスオートマトンを導入する準備として，有限状態オートマトンを定義する．**非決定的有限状態オートマトン** M は $(\Omega, Q, Q^{\mathrm{init}}, Q^{\mathrm{fin}}, \delta)$ の組である．ここで，Ω はアルファベット，Q は**状態**の有限集合，$Q^{\mathrm{init}} \subseteq Q$ は**初期状態**の集合，$Q^{\mathrm{fin}} \subseteq Q$ は**終了状態**の集合，δ は $Q \times \Omega$ から Q の巾集合への**遷移関数**である[12]．集合 Q^{init} が 1 つしか要素を持たず，遷移関数 δ の返す状態集合も 1 つしか要素を持たないなら M は**決定的**である．

パスオートマトンは，正規木文法から構築される非決定的有限状態オートマトンである．正規木文法 $G = (N, \Sigma, S, P)$ からパスオートマトン $M[G]$ を構築するには，すべての非終端記号（すなわち N）を終了状態として扱う．ほかに，開始状態 q^{ini} を導入する．$M[G]$ は以下のように定義される．

$$M[G] = (\Sigma, N \cup \{q^{\mathrm{ini}}\}, \{q^{\mathrm{ini}}\}, N, \delta),$$

ここで δ は $(N \cup \{q^{\mathrm{ini}}\}) \times \Sigma$ から $N \cup \{q^{\mathrm{ini}}\}$ の巾集合への遷移関数であり，

$$\begin{aligned}\delta(x, a) = \{x' \mid &\text{ある生成規則 } x \to r \in P \text{に対し}\\ &a[x'] \text{が } r \text{中に出現する}\}\\ \cup \{x' \mid &x = q^{\mathrm{ini}}, a[x'] \in S\}\end{aligned}$$

を満たすものとする（a は Σ に属する名前である）．

例 5 例 1 のスキーマ G_1 に対するパスオートマトン（**図 1**）は，

$$M[G_1] = (\Sigma_1, N_1 \cup \{q^{\mathrm{ini}}\}, \{q^{\mathrm{ini}}\}, N_1, \delta)$$

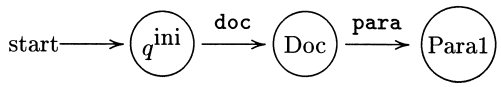

図 1 パスオートマトン $M[G_1]$
Fig. 1 Path automaton $M[G_1]$.

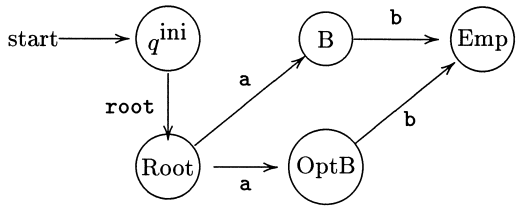

図 2 パスオートマトン $M[G_2]$
Fig. 2 Path automaton $M[G_2]$.

である．ここで

$$\begin{aligned}&\Sigma_1 = \{\texttt{doc}, \texttt{para}\},\\ &N_1 = \{\mathrm{Doc}, \mathrm{Para1}\},\\ &\delta(q^{\mathrm{ini}}, \texttt{doc}) = \{\mathrm{Doc}\},\\ &\delta(\mathrm{Doc}, \texttt{para}) = \{\mathrm{Para1}\}.\end{aligned}$$

G_1 は単一型木文法であり，$M[G_1]$ は決定的であることに注意． □

例 6 例 3 のスキーマ G_2 に対するパスオートマトン（**図 2**）は

$$M[G_2] = (\Sigma_2, N_2 \cup \{q^{\mathrm{ini}}\}, \{q^{\mathrm{ini}}\}, N_2, \delta)$$

である．ここで

$$\begin{aligned}&\Sigma_2 = \{\texttt{root}, \texttt{a}, \texttt{b}\},\\ &N_2 = \{\mathrm{Root},\ \mathrm{B},\ \mathrm{OptB},\ \mathrm{Emp}\}\\ &\delta(q^{\mathrm{ini}}, \texttt{root}) = \{\mathrm{Root}\},\\ &\delta(\mathrm{Root}, \texttt{a}) = \{\mathrm{B}, \mathrm{OptB}\},\\ &\delta(\mathrm{B}, \texttt{b}) = \{\mathrm{Emp}\},\\ &\delta(\mathrm{OptB}, \texttt{b}) = \{\mathrm{Emp}\}.\end{aligned}$$

G_2 は正規木文法であり，$M[G_2]$ は非決定的であることに注意． □

G が単一型木文法であるとき，またそのときに限り，G から構築されるパスオートマトン $M[G]$ は決定的であることを示す．G が単一型文法であれば，条件 (1) から $\delta(q^{\mathrm{ini}}, e)$ は 1 つしか要素を持たない．また，条件 (2) から，$\delta(x, e)$（ここで $x \in N$）は 1 つしか要素を持たない．したがって，$M[G]$ は決定的である．逆に，パスオートマトン $M[G]$ が決定的であれば，条件 (1) と (2) が成立するので，G は単一型木文法である．

3.2 計 算 量

パスオートマトン構築には，正規木文法の大きさに比例する計算時間を必要とする．正確には，生成規則の右辺に現れる式 $a[x']$ の個数に比例する．この形の式 1 つに対して，遷移が 1 つ構築されるからである．

次に，パスオートマトン $M[G]$ の実行について考察する．長さ n のパス 1 つに対してパスオートマトン $M[G]$ を実行するには，最悪の場合でも，N（非終端記号）の濃度 $|N|$ の二乗と n との積に比例する計算時間で十分である．1 つの文書に含まれるすべてのパスに対して $M[G]$ を実行するには，最悪の場合でも，$|N|^2$ と文書中の要素数との積に比例する計算時間で十分である．

4. 型情報の付与

本章では，各要素までのパスに対してパスオートマトンを実行することによって型情報を付与する方法を示す．まず，スキーマが単一型木文法であって，文書がスキーマに照らして妥当であると保証されている場合を扱う．次に，スキーマが正規木文法であって，妥当性が保証されている場合を扱う．最後に，妥当性が保証されていない場合を扱う．

文書中のある要素 e を考える．ルート要素から要素 e までのパスに対して，パスオートマトン $M[G]$ を実行すると，いくつかの状態からなる集合が得られる．この集合を e^G で表すことにする．正確には，e^G は次のように定義される．

$$
\begin{aligned}
X_0 &= \{q^{\mathrm{ini}}\},\\
X_1 &= \bigcup_{q\in X_0} \delta(q, a_1),\\
X_2 &= \bigcup_{q\in X_1} \delta(q, a_2),\\
&\cdots\\
X_n &= \bigcup_{q\in X_{n-1}} \delta(q, a_n),\\
e^G &= \bigcup_{q\in X_n} \delta(q, a).
\end{aligned}
$$

ここで，$a_1, a_2, \ldots, a_n, a$ は，このパスに現れる要素のラベルである（a_1 はルート要素のラベル，a_n は要素 e の親要素のラベル，a は要素 e のラベル）．$X_0, X_1, \ldots, X_n$ は状態の集合である．

4.1 単一型文法かつ妥当な場合

スキーマ G が単一型文法の場合は，パスオートマトン $M[G]$ は決定的であるから，e^G は必ず 1 つしか状態を持たない．

文書がスキーマに照らして妥当であると保証されている場合には，解釈はただ 1 つ存在する．したがって，e^G に含まれる非終端記号をそのまま型として使用することができる．

例 7 例 2 で示した文書の 2 つの `para` に対して型情報を付与することを考える．これらの要素へのパスは `/doc/para` であり，例 5 のパスオートマトン $M[G_1]$ を実行して得られる状態は Para1 である．したがって，2 つの `para` 要素の型として Para1 を使用することができる． □

4.2 正規木文法かつ妥当な場合

スキーマ G が正規木文法であって単一型文法でない場合は，パスオートマトン $M[G]$ は非決定的である．したがって，e^G が複数の状態を含むことがありうる．この場合は，パスオートマトンの実行だけによって e の型を一意に決めることはできない．この場合に対処するために，部分的な検証と不確定を表す型を用いる．

4.2.1 部分的な検証

e^G に含まれる状態 x（これはスキーマ G の非終端記号）に対して，e の内容を検証することによって，e の型を一意に決められることがある．このような検証を，部分的な検証と呼ぶ．

例 8 例 4 で示した文書の 2 番目の `a` 要素に対して，型情報を付与することを考える．例 6 のパスオートマトン $M[G_2]$ の実行によって得られる状態は，OptB と B の両方である．しかし，2 番目の `a` は空要素なので，OptB に照らして妥当だが，B に照らして妥当ではない．したがって，OptB が型であると確定することができる． □

部分的な検証がとくに有効なのは，単純型に対する場合である．1 個以上の子要素を含む内容は，明らかに単純型には適合しない．文字列だけからなる内容を，単純型と照合することは容易に実現できる．一方，複合型の場合には，文書全体をスキーマに照らして検証するときと同じ処理が必要になるという問題がある．

4.2.2 不確定を表す型

XQuery の仕様は，文書の一部について検証が失敗した場合などのために，すべてを許容する型として `xs:anyType`（要素の場合）を許している．

e^G が複数の状態を含んでおり，部分的な検証によってその 1 つに確定することができない場合には，`xs:anyType` を型として用いる．単純型については部分的検証によって一意に確定することがほとんどなので，`xs:anyType` が用いられるのは複合型の場合にほぼ限られる．

4.3 妥当であるという保証がない場合

妥当であるという保証がない場合における型情報の付与を考える．妥当でない場合には，そもそも解釈が存在しない．妥当でない場合に型情報を付与するのは不適切であり，検証によって妥当性を確認すべきであるという考え方もある．

しかし，実用的には，たとえ妥当でなくても型情報をできるだけ付与したいという要求がある．文書のごく一部が誤っているから，もしくは現時点では完成できないからといって，XQueryでの取扱いを禁止するのは現実的ではない．たとえば，スキーマでは許されていない名前空間の要素・属性が使われているが，これらの要素・属性さえ無視すれば問題なく動作するという文書は多い．

前節で導入した，部分的な検証と不確定を表す型とを利用すれば，文書全体が妥当でない場合にも型情報を付与することができる．要素 e が 1 つ以上の子要素を持つ場合は，`xs:anyType` を付与する．要素 e が文字列だけを内容として持つ場合は，まず e^G を計算する．e^G に含まれる単純型のうち 1 つだけにこの文字列がマッチするなら，これを e の型情報として用いる．複数の単純型にマッチするなら `xs:anyType` を用いる．

5. 属性・単純型・混在内容モデルの扱い

前章までは要素だけを扱っていたが，本章ではスキーマ言語 RELAX NG に限定して属性・単純型・混在内容モデルを扱うように拡張する．これらを含むスキーマからどのようにパスオートマトンを構築し，どのように型情報を付与するかを示す．

5.1 属　　性

RELAX NG においては，属性と要素はどちらも 1 つの式の中に記述される．両者はほとんど同等に扱われる．要素 `foo` と属性 `foo` の一方を選択することを表す内容モデルを持つ生成規則（RELAX NG の define 文）に示す．

```
<define name="x">
  <choice>
    <element name="foo">
      <ref name="Bar"/>
    </element>
    <attribute name="foo">
      <ref name="Bar"/>
    </attribute>
  </choice>
</define>
```

スキーマからパスオートマトンを構築するときは，属性と要素を区別することなく扱う．上記の define 文は，以下の生成規則として扱う．

$$x \quad \rightarrow \quad \texttt{foo}[\mathrm{Bar}] \mid \texttt{@foo}[\mathrm{Bar}]$$

パスオートマトンの構築に関する限り，要素と属性の唯一の違いは，要素の名前と属性の名前とは区別されるという点だけである．ここでは属性名 `foo` の先頭に `@` を付け加えて区別している．状態遷移として，状態 x から状態 Bar への `foo` による遷移と，同じく状態 x から状態 Bar への `@foo` による遷移が作成される．

パスオートマトンの実行においても，属性の扱いは要素と同じである．ただし，パスオートマトンによって状態が一意に確定できない場合には，不確定を表す型として `xs:anySimpleType` を用いる（要素のときは `xs:anyType` を用いた）．`xs:anySimpleType` は，XQuery が提供している型である．

5.2 単　純　型

`data` と `value` は，単純型を指定するための機構である．`data` は単純型を指定するだけだが，`value` は単純型と値の両方を指定する．たとえば，文字列 `"01"` と文字列 `"2"` は `<data type="xsd:int"/>` にマッチする．文字列 `"01"` は `<value type="xsd:int">1</data>` とマッチするが，文字列 `"2"` はマッチしない．

`data` が指定されるたびに，それに対応する非終端記号 Data_i を導入する．同様に，`value` が指定されるたびに，それに対応する非終端記号 Value_j を導入する．例として以下の define 文を考える．

```
<define name="x">
  <element name="foo">
    <data type="xsd:integer"/>
  </element>
  <element name="bar">
    <data type="xsd:float"/>
  </element>
  <element name="baz">
    <value type="xsd:integer">1</value>
  </element>
</define>
```

この define 文は，以下の生成規則として扱う．

$$x \quad \rightarrow \quad \texttt{foo}[\mathrm{Data}_1]\texttt{bar}[\mathrm{Data}_2]\texttt{baz}[\mathrm{Value}_1]$$

ここで，Data_1 と Data_2 は，それぞれ 1 番目と 2 番目の `data` 要素に対応する非終端記号であり，Value_1 は `value` 要素に対応する非終端記号である．状態遷

移として，状態 x から状態 Data_1 への foo による遷移，状態 x から状態 Data_2 への bar による遷移，状態 x から状態 Value_1 への baz による遷移が作成される．

5.3 混在内容モデル

text は，文字列を指定するための RELAX NG の機構である．子要素に関する制約に data や value を連結することはできないが，text を連結することはできる．text は，DTD でいう混在内容モデル（すなわち #PCDATA）に相当する．

text は，非終端記号 Text であると見なす．例として以下の define 文を考える．

```
<define name="x">
  <element name="foo">
    <text/>
  </element>
</define>
```

この define 文は，以下の生成規則として扱う．

$$x \quad \rightarrow \quad \ldots\texttt{foo}[\mathrm{Text}]$$

状態遷移としては，状態 x から状態 Text への foo による遷移が作成される．

6. 実　　装

本章では，RELAX NG スキーマからパスオートマトンを構築するプログラムの実装について述べる．

このプログラムは，RELAX NG のための検証器 Jing を利用して作成されている．Jing のスキーマ内部形式は，2 章で示した正規木文法とほぼ同等である．Jing によって RELAX NG スキーマを内部形式に変換したのち，この内部形式からパスオートマトンを構築する．

以下に，RELAX NG スキーマの例を示す．このスキーマは，例 3 の正規木文法 G_2 に，データ型として xsd:int（整数型）を指定したものである．

```
<grammar
   xmlns="http://relaxng.org/ns/structure/1.0"
  datatypeLibrary=
    "http://www.w3.org/2001/XMLSchema-datatypes">
  <start>
    <element name="root">
      <ref name="Root"/>
    </element>
  </start>
  <define name="Root">
    <choice>
      <group>
        <element name="a">
          <ref name="OptB"/>
        </element>
        <element name="a">
          <ref name="B"/>
        </element>
      </group>
      <group>
        <element name="a">
          <ref name="b"/>
        </element>
        <element name="a">
          <ref name="OptB"/>
        </element>
      </group>
    </choice>
  </define>
  <define name="B">
    <element name="b">
      <data type="int"/>
    </element>
  </define>
  <define name="OptB">
    <optional>
      <element name="b">
        <data type="int"/>
      </element>
    </optional>
  </define>
</grammar>
```

以下に上のスキーマから生成したパスオートマトンを XML 文書の形で示す．t，s，d はそれぞれ transition，source，destination の頭文字である．

```
<pathAutomaton
  xmlns:rng=
    "http://relaxng.org/ns/structure/1.0"
  start="#start">
<t s="#start" d="Root">
  <rng:element>
    <rng:name ns="">root</rng:name>
  </rng:element>
</t>
<t s="Root" d="OptB">
  <rng:element>
    <rng:name ns="">a</rng:name>
```

```
  </rng:element>
</t>
<t s="Root" d="B">
  <rng:element>
    <rng:name ns="">a</rng:name>
  </rng:element>
</t>
<t s="OptB">
  <rng:element>
    <rng:name ns="">b</rng:name>
  </rng:element>
  <d>
    <rng:data type="int"
      datatypeLibrary=
"http://www.w3.org/2001/XMLSchema-datatypes"/>
  </d>
</t>
<t s="B">
  <rng:element>
    <rng:name ns="">b</rng:name>
  </rng:element>
  <d>
    <rng:data type="int"
      datatypeLibrary=
"http://www.w3.org/2001/XMLSchema-datatypes"/>
  </d>
</t>
</pathAutomaton>
```

このパスオートマトンを用いて以下の文書に型情報を付与することを考える．

```
<root>
  <a><b>3</b></a>
  <a><b>0</b></a>
</root>
```

要素 <b> へのパスは，/root/a/b である．このオートマトンを実行すると，以下の状態（単純型）が型の候補として得られる．

```
<rng:data type="int"
  datatypeLibrary=
"http://www.w3.org/2001/XMLSchema-datatypes"/>
```

文字列 "3" と "0" はこの単純型（xsd:int）に属するので，これが要素 <b> の型であると決定できる．

7. 議　　論

XML1.0 勧告における XML 文書は，要素と属性と文字列からなる単純な木である．型といえるものは，XML 1.0 にはほとんど存在しない（整数型さえも存在しない）．この単純さは，XML をプログラミング言語独立にすること，XML パーサの実装を容易にすることに貢献した．

一方，XML を扱うアプリケーションプログラムは，型を必要とすることが多い．従来は，型を導入することはプログラマの責任であった．すなわち，型を持つデータと XML との間の変換プログラムを，プログラマが実装していた．この作業はプログラマにとっては大きな負担であり，“XML Is Too Hard For Programmers” といわれるようになった[13]．

最近は，スキーマを型情報として用いることがさかんに試みられている．XML プログラミング言語（XDuce[11]など），XML データバインディングツール（JAXB など）などはこのような試みに相当する．XQuery では，スキーマに書かれた型情報を検証によって文書に付与するという方式[14]をとっている．

スキーマから単純型を得て文書に付与することの有効性は明らかである．単純型を利用しない場合は，<a>3 という要素の内容は単なる文字列にすぎない．スキーマから整数型を得ることができれば 3 という整数として扱うことができる．単純型をスキーマから得ることによって，プログラミングが容易になる・プログラムの信頼性が向上するというメリットがある．スキーマを用いたほとんどの XML プログラミング言語や XML データバインディングツールは，単純型を利用している．

一方，スキーマから複合型を得て文書に付与することの有効性については，意見が分かれる．関数型言語 XDuce は，強力なパターンマッチ機構を備えているが，スキーマに書かれた複合型によってパターンマッチの動作が変わることはなく，解釈の一意性を要求することもない．一方，XDuce は正規木言語のブール演算に基づく型推論を備えている．ブール演算についての閉包性を必要とする型推論を行う以上，解釈の一意性を保証する単一型木文法を利用することはできないともいえる．一方，データバインディングツールは，文書の解釈を 1 つ構築し，スキーマに書かれた複合型を積極的に利用する．たとえば，JAXB や Relaxer では，Java クラスの生成に複合型を用いる．複合型を決めずに要素を参照することはいっさいできない．最後に，XQuery は，スキーマに書かれた複合型を積極的に利用するわけでも，まったく用いないわけでもない．複合型を利用する機構（複合型を指定して要素や属性とのマッチを判定する機構など）を備えてはいるが，それを用いない検索式も数多く存在する．

本研究の方式では，スキーマからパスオートマトンを作成し，それを各パスに対して実行することによって型情報を付与する．型を一意に確定できない場合は，部分的な検証（文字列と単純型とのマッチ）によって補う．それでも確定できない場合は，未確定を表す型を用いる．

本方式では，単純型の付与はほとんどの場合に可能だが，複合型の付与はそうではない（詳しくは 4 章を参照）．一方，本方式には，検証という重い処理を省けるという利点，RELAX NG に適用可能であるという利点がある．

4 章で示したように，文書の妥当性が保証されている場合には，保証されていない場合と比べて的確な型付けができる．検証を行うことなしに，妥当性を保証できるのは次のような文書である．

- XDuce のような静的な型付けを行うシステムによって作成される文書
- JAXB の on-demand validation を通った Java オブジェクトから作成される文書
- XML 以外のスキーマ（たとえば RDB スキーマ）に従うデータから自動生成された文書

現実の利用において，このような文書がどれだけ存在するかを調べることは，今後の課題である．

8. 結　　論

本論文では，型情報の付与を検証から分離する方式を提案した．この方式は，スキーマからパスオートマトンを構築する処理，パスオートマトンを実行して型情報を付与する処理からなる．スキーマとして単一型木文法だけを許し，文書の妥当性が保証されている場合は，本方式によって単純型と複合型のどちらも確定することができる．スキーマとして任意の正規木文法を許す場合または文書の妥当性が保証されていない場合は，パスオートマトンの実行だけでは不十分であるが，部分的な検証を行うことおよび不確定を表す型を利用することによって補うことができる．

XQuery ではスキーマから型情報を文書に付与するという方式[14]を採用しているが，7 章で議論したようにこの方式の有効性はまだ明らかではない．本方式はパスオートマトンをスキーマから構築しているが，スキーマ以外の手段によってパスオートマトンを構築することを今後は検討していく必要がある．

参 考 文 献

1) Boag, S., Chamberlin, D., Fernández, M., Florescu, D., Robie, J. and Siméon, J.: XQuery 1.0: An XML Query Language, W3C Working Draft (2003).
2) Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E. and Yergeau, F.: Extensible Markup Language (XML) 1.0 (3rd Edition), W3C Recommendation (2004).
3) Fallside, D.C.: XML Schema Part 0: Primer, W3C Recommendation (2001).
4) Thompson, H.S., Beech, D., Maloney, M. and Mendelsohn, N.: XML Schema Part 1: Structures, W3C Recommendation (2001).
5) Biron, P.V. and Malhotra, A.: XML Schema Part 2: Datatypes, W3C Recommendation (2001).
6) Clark, J. and Murata, M.: RELAX NG Specification (2001).
7) Mignet, L., Barbosa, D. and Veltri, P.: The XML Web: a First Study, *Int'l World Wide Web Conf.* (WWW) (2003).
8) Nicol, M. and John, J.: XML Parsing: a threat to database performance, *ACM CIKM* (2003).
9) 川口耕介：XML & Java の高速化技法，*XML World*, Vol.8, No.1, pp.138–150 (2004).
10) Murata, M., Lee, D. and Mani, M.: Taxonomy of XML Schema Languages using Formal Language Theory, *Extreme Markup Languages*, Montreal, Canada (2001). http://www.cs.ucla.edu/~dongwon/paper/
11) Hosoya, H. and Pierce, B.C.: XDuce: A statically typed XML processing language, *ACM Trans. Internet Technology* (*TOIT*), Vol.3, No.2, pp.117–148 (2003).
12) Hopcroft, J.E. and Ullman, J.D.: *Introduction to Automata Theory, Language and Computation*, Addison-Wesley (1979).
13) Bray, T.: XML Is Too Hard For Programmers (2003). http://www.tbray.org/ongoing/When/200x/2003/03/16/XML-Prog
14) Simeon, J. and Wadler, P.: The Essence of XML, *Principles of Programming Languages*, New Orleans (2003).

(平成 15 年 9 月 25 日受付)
(平成 16 年 1 月 19 日採録)

（担当編集委員　石川 博，市川 哲彦，原 隆浩，
佐藤 聡，土田 正士）

村田　　真（正会員）

昭和 35 年生．昭和 57 年京都大学理学部卒業．同年富士ファコム制御（株）入社．昭和 61 年富士ゼロックス（株）入社．平成 5 年から 7 年まで米国ゼロックス社の Webster Research Center に滞在．平成 7 年から富士ゼロックス情報システム（株）に出向．平成 12 年から日本 IBM（株）東京基礎研究所特別研究員および国際大学研究所特任研究員．W3C XML ワーキンググループのメンバとして XML 勧告の制定に貢献．編著書に，『XML 入門』（日本経済新聞社）．インターネットコンファレンス 98 論文賞．