情報処理学会論文誌 Vol. 49 No. 7 2604–2613 (July 2008)

繰返し構造認識によるXMLパーサ高速化技術

太田智也[†1] 西山博泰[†1]
田村清朗[†1] 宗近秀生[†1]

データ交換のための標準的な形式としてXMLが広く利用されている．XMLはテキスト形式で表現された柔軟性の高いフォーマットであるが，一方で，データ処理のオーバヘッドが高いという問題をかかえている．そこで，本稿では，あらかじめ解析対象のXML文書に頻出する構造を学習し，文書の解析時には学習した構造に対して予測的に解析処理を適用することで処理時間の短縮を図る高速化手法を提案する．本手法を適用することにより，解析対象のXML文書と類似の構造を持つXML文書を学習文書として与えた場合，SAXパーサの処理性能が，平均31%，最大67%向上することを確認した．

A High Performance XML Parser Using Repetition Pattern Recognition

Tomoya Ohta,[†1] Hiroyasu Nishiyama,[†1]
Seiro Tamura[†1] and Hideo Munechika[†1]

XML is widely used as a standard format for data exchange. XML has high flexibility of representation because of its adoption of a text format. However, this flexibility costs higher data processing overhead than ordinaly data formats. In this paper, we propose a high performance XML processing method using repetition pattern recognition of XML documents. In the method, first, training XML documents are pre-analyzed in order to detect frequently appearing constructs in the document. XML parser uses the result of the pre-analyzed to make its parsing faster with speculative input matching. The results of experiments shows that the introduced method improve the performance of XML parsing up to 67% (31% on average) compared with a conventional SAX parser under the condition that the target XML documents are similar to the pre-analyzed XML documents.

1. はじめに

従来プログラム間のデータ交換には固定形式のデータが利用されていた．このような固定的データ形式は柔軟性が低く，データ形式やプログラムの更新，ネットワークを介したデータのやりとりで問題を生じることが多々あった．そこで，データ交換のための標準的な形式として，XML（Extensible Markup Language）[1)]が提案され，広く利用されている．

XMLは固定的な構造を持たない半構造データであり，タグおよびデータから構成されたテキストデータによって表現される．タグによってデータの内容を識別することが可能となるため，従来の固定的なフォーマットに比べてデータ表現の柔軟性が高くなっている．また，タグのネストによって階層的なデータ構造を表現することも可能である．このような点から，XMLはネットワーク経由でデータ交換やサービス呼び出しを行うための軽量プロトコルであるSOAP[2)]など，様々な分野で広く応用されている．

XMLで表現されたデータはテキスト形式で表現されるため，XML形式で記述されたデータを処理するためには，タグとデータを分離し，XMLデータの構造を解析する必要がある．このような処理を行うソフトウェアをXMLパーサと呼ぶ．XMLデータの構造解析手法としては，DOM（Document Object Model）[3)]による手法と，SAX（Simple API for XML）[4)]による手法が標準的に利用されている．

DOMは入力したXMLデータをそれに対応した木状データ構造へ変換し，それに対して操作を行うモデルを採用している．一方，SAXは入力XMLデータの構文各要素に対してイベントを定義し，入力した各要素に対応するユーザ定義のイベントハンドラを呼び出すことにより処理を行う，イベント駆動のモデルを採用している．一般に，DOMパーサは内部表現への変換をともなうため，SAXパーサと比較して，時間/空間的な効率が低い．一方，SAXパーサではXMLデータの階層的な構造を直接認識しないため，XML文書の構造が複雑な場合に処理が煩雑になるという問題がある．

本稿では，SAX型のイベント駆動パーサを対象としたXMLデータの解析処理の高速化手法を提案する．本稿の手法では，あらかじめ解析対象の文書に頻出する構造を学習し，文書の解析時には学習した構造に対して予測的に解析処理を適用することで処理時間の短縮を図る．

†1 株式会社日立製作所
Hitachi Ltd.

以下，2 章では，XML 文書の処理方式について述べ，3 章では提案手法を述べる．4 章では評価結果について述べる．5 章では関連研究について述べる．最後に 6 章で結論を述べる．

2. XML 文書の解析手法

本稿の対象とする SAX パーサによる XML 文書の解析モデルでは，XML の各構文要素に対してイベントを定義する．XML 処理を記述するユーザは，各イベントに対して実行すべき処理を記述したハンドラを定義し，SAX パーサに与える．SAX パーサが入力した構文要素に対応したイベントハンドラを呼び出すことにより，XML データの処理が行われる．

典型的な SAX パーサでの XML 文書の解析手順は次となる．

1. 入力 XML 文書をファイルなどから読み込む．
2. 入力文字列をトークンへ分解する．このとき，タグ名などが，XML の規格で定められている範囲の文字を使用しているか否か合わせてチェックする．
3. イベントハンドラの引数となるデータを構成する．
4. イベントハンドラを呼び出す．

このとき，もし，次に読み込まれるトークンが予測可能な場合は，ステップ 2 の処理は予測トークンと一致するかを検査する単純なマッチング処理とすることが可能となる．また，ステップ 3 の引数データの構成は，解析時にオンデマンドで構成するのではなく，予測トークンに合わせて，あらかじめ用意しておくことが可能となる．これらにより，文書解析処理の高速化が期待できる．

よって，XML パーサに対して，XML 文書の頻出パターンを登録し，そのパターンに一致する間は予測的に XML 文書を解析することにより，性能向上が期待できる．

3. 提 案 手 法

3.1 構　　成

提案手法の構成を**図 1** に示す．改良版 XML パーサにおける文書の解析は，学習フェーズと，実行フェーズの 2 つに分かれる．学習フェーズでは，まず，頻出構造認識部により，与えられた XML 文書（学習用 XML 文書）を解析して頻出パターンを検出する．次に，最適化スキャナ生成部で検出された頻出パターンのそれぞれに対して，そのパターンと入力トークンとの一致を解析する専用のスキャナである最適化スキャナを生成する．実行フェーズでは，解析処理に先立ち，最適化スキャナをロードする．解析処理では，まず，通常のスキャナによる解析を行う．スキャナは，現在読み取っているトークンが最適化スキャナで実

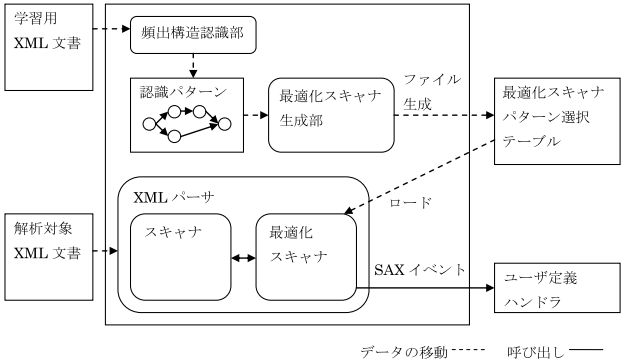


図 1　改良版 XML パーサの構成図
Fig. 1　System organization of proposed XML parser.

装されているパターンの先頭であるか否かを調べる．もし，最適化スキャナで実装されているパターンの先頭である場合は，処理を最適化スキャナに移す．最適化スキャナでは予測的に XML 文書を解析するため，XML 解析処理を高速化できる．もし，最適化スキャナの処理中に入力が実装されているパターンと異なった場合は，処理を通常のスキャナに戻すことで，後続のトークンの処理を行う．なお，本システムでは，最適化スキャナは Java プログラムとして生成し，バイトコードにコンパイルして保持する．

以下，頻出構造認識部，最適化スキャナ生成部，スキャナについて述べる．

3.2 頻出構造認識部

学習用 XML 文書からコンパクトな認識パターン情報を生成するためには，入力データ中に繰り返し現れるパターンを識別する必要がある．本研究では，文法変換による圧縮手法を利用して，入力データを文脈自由文法による表現に変換し，そこから頻出するパターンを認識することによりマッチング用のオートマトンを生成する．

3.2.1 文法表現への変換

文法変換による圧縮手法とは，与えられた入力データに対して，それを一意に導出する文脈自由文法を構成し，生成された文法を符号化することにより与えられた入力データを圧縮する可逆データ圧縮手法である[5]．本研究では，文法変換による圧縮手法の 1 つである，Nevill-Manning による SEQUITUR[6] アルゴリズムを用いる．SEQUITUR アルゴリズムでは以下の条件を満たすように，入力から文法規則を生成する．

1. 同一の隣接シンボルの組（digram）は文法中で 1 度しか出現しない．

```
<items>
 <item>
  <name>item0</name><price>100</price>
 </item>
 <item>
  <name>item1</name><price>150</price>
 </item>
 <item>
  <name>item2</name><price> 50</price>
 </item>
 <item>
  <name>item3</name><price>800</price>
 </item>
</items>
```

図 2　XML 文書例
Fig. 2　An example of XML document.

2.　開始規則以外のすべての生成規則の利用回数は 2 回以上である．

後者の条件により，入力シンボル列に複数回現れるパターンは，生成規則として括りだされる．そのため，生成された文法中の頻出する生成規則が，頻出パターンとなる．

SEQUITUR アルゴリズムにより生成された文法は，元の入力シンボル列の階層的構造を表現するという特徴を持つ．このため，SEQUITUR アルゴリズムの入力となるシンボルを基本的にタグの単位とすることにより，XML 文書の構造を自然に表現することができる．なお，コンテンツ部分などの基本的に異なる値が現れる部分については，任意の文字列を表すシンボル “#” に置き換えて SEQUITUR アルゴリズムの入力とする．たとえば，

“<a> data ”

という XML 文書は，

“<a>”，“#”，“”，“#”，“”，“#”，“”

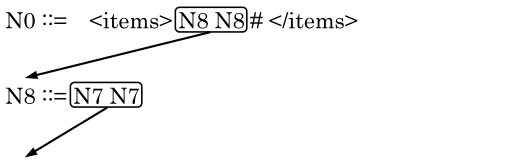
の 7 つのトークンに分解する．なお，ここで空白，改行文字もシンボル “#” で表していることに注意されたい．

図 2 の XML データに対する SEQUITUR アルゴリズムの適用例を**図 3** に示す．図の入力列は次の入力シンボルを，また，生成プロセス列は入力シンボルを読み取る前の生成規則を示している．まず，Step 1 では，初期状態として空の生成規則，「N0 ::=」と，最初の入力シンボル「<items>」が与えられる．この時点で適用可能な条件は存在しないため，N0 に

Step	入力	生成プロセス
1	<items>	N0 ::=
2	#	N0 ::= <items>
3	<item>	N0 ::= <items>#
:	:	:
k	<item>	N0 ::= <items> # <item> # <name> … </item> #
k1	#	N0 ::= <items> N1 # <name> … </item> N1 N1 :: = # <item>
k2	<name>	N0::= <items> N2 <name> … </item> N2 N1::= # <item> N2 ::= N1 #
k3	<name>	N0 ::= <items> N2 <name> … </item> N2 N2 ::= # <item>
:	:	:
L	$	N0 ::= <items> N8 N8 # </items> N8 ::= N7 N7 N7 ::= # <item> # <name> # </name> # <price> # </price> # </item>

図 3　文法生成の過程
Fig. 3　Process of rule generation.

入力を結合した生成規則「N0 ::= <items>」が構成される．その後，2 つ目の「<item>」が入力される（Step k）までは，入力はそのまま N0 に追加される．Step k においては，生成規則 N0 にシンボル「<item>」を結合すると，digram {#, <item>} が文法中に 2 度出現するため，条件 1 に反する．よって，新たな生成規則「N1 ::= # <item>」を作成し，digram を置換する（Step k1）．次の入力「#」を処理すると，同様に digram{N1, #} が文法中に出現するため，新たな生成規則「N2 ::= N1 #」で置き換える（Step k2）．置換後の文法では，生成規則 N1 の利用回数が 1 度となり，条件 2 に反する．よって，生成規則 N1 を参照元の生成規則 N2 に展開し，生成規則「N2 ::= # <item> #」を得る．この過程を繰り返すことにより，最終的に Step L に示す生成規則が得られる．なお，「$」は最後



(a) 適用前

N0 ::= <items> N8 # </items>
N8 ::= N7*
N7 ::= # <item> # <name> # </name> ... </item>

(b) 適用後

図 4　繰返し構造の認識
Fig. 4　Recognition of repetition structure.

の入力を表している．

SEQUITUR アルゴリズムによって得られた文法から，入力中に繰り返して出現するシーケンス「#, <item>, . . . , </item>」が 1 つにまとめられ，入力データの圧縮が行われていることが分かる．

3.2.2　繰返し構造の認識

前項で示した手法で得られる文法は，学習用 XML 文書の値部分は任意文字列となっているが，XML 文書構造としては，まったく等価である．しかし，ここでの目的は頻出パターンを認識することであり，そのためには，学習文書に現れる繰返し回数に固定されたパターンでは都合が悪い．そのため，学習 XML 文法が繰返し構造を持つ場合，この文法を繰返しを含む文法へ変換することにより，より広範囲にマッチングする文法を生成する．

3.2.1 項で示した手法により得られる文法では，入力データにおける繰返しパターンは，**図 4** (a) の例に示すように，同一記号からなる digram を右辺に持つ生成規則の階層的な参照によって表現される．よって，繰返し構造の認識処理は次となる．

繰返し構造の認識：

1. 文法を開始記号から再帰的にたどる．
2. 生成規則が同一記号からなる digram を右辺に持つ生成規則「$\alpha ::= \beta\ \beta$」である場合，右辺を繰返し「$\alpha ::= \beta*$」に変換する．
3. 生成規則中に出現する同一記号からなる digram「$\ldots \alpha\ \alpha \ldots$」について，$\alpha$ の生成規則が繰返し「$\alpha ::= \beta*$」であれば，digram を α で置換し，「$\ldots \alpha \ldots$」とする．

例題の入力に対して繰返し構造の認識を行った結果生成される文法を図 4 (b) に示す．繰返し構造の認識の結果，生成規則 N8 の右辺が繰返しに変換される．

次に，繰返し構造がネストする場合を考える．この場合，ネスト内側の繰返し構造の回数が異なると，それぞれが異なる生成規則として認識され，上記の処理のみでは正しく繰返し構造を認識できない．そのため，SEQUITUR アルゴリズム適用段階で，4 回以上連続するシンボルは 4 個目までを認識し，5 個目以降は無視する規則を追加する．これにより，生成される文法は繰返し回数において学習用 XML 文書と異なるものになるが，その後適用される繰返し認識ステップにおいて，繰返し構造と認識される．

図 5 に例を示す．同図 (a) に示す文書を学習する場合，規則の追加がない場合は，同図 (b) に示す文法が構成されていた．この場合，生成規則 N2 により，要素 B の繰返しが認識されていた．新たな規則を追加した場合，同図 (c) に示す文法が構成される．この場合，生成規則 S により，N3，つまり，要素 B の繰返しを含んだ，要素 A の繰返しが認識される．

3.2.3　生成規則の変形

改良版 XML パーサの実行フェーズでは，学習用 XML 文書から得られたパターンと解析対象の入力文書とのマッチングを行う．このとき，すべての要素において，パターンとのマッチングを行った場合，パターン適用の機会は増大するが，マッチング処理の呼び出し回数も増加し，オーバヘッドも増大する．そこで，学習パターンの先頭は，開始タグに対応するトークンに限定し，開始タグの処理のタイミングでのみ，マッチング処理を呼び出すことにする．これは，XML 文書では，開始タグと終了タグが必ず組で現れること，任意の文字列を表すトークンはマッチングの先頭としては使えないことによる．

学習パターンは，学習用 XML 文書から生成された文法の各生成規則から生成する．前項までの手順により生成された文法の生成規則は，規則の先頭が開始タグである保証はない．そこで，ここでは，学習された文法の各生成規則に対して，生成規則が開始タグで始まるように変形した規則を作成し，その規則から学習結果のパターンを生成する．変形処理は，繰返しを表す生成規則と通常の生成規則で別の処理となる．繰返し規則に対する処理では，ピーリングを適用する．ピーリングでは，繰返し規則の開始位置をずらした生成規則を生成する．たとえば，構造認識後の文法（図 4 (b)，**図 6** (a)）において，N7 の生成規則の右辺の先頭の “#” は，パターンの先頭として無効なシンボルであり，このままでは，繰返し規則 N8 は，マッチングの対象とならなくなる．ピーリングを適用すると，図 6 (b) の規則が生成され，新たな繰返し規則 N11 では，規則の先頭が “<item>” となるため，マッチング

```
<A>
  <B>#</B>…<B>#</B>   // 繰り返し 4 回
</A>
<A>
  <B>#</B>…<B>#</B>   // 繰り返し 5 回
</A>
```

(a) 学習 XML 文書

```
S ::= N3 N4 N3 N1 N4
N1 ::= <B> # </B>
N2 ::= N1 N1
N3 ::= <A> # N2 N2
N4 ::= # </A> #
```

(b) 打ち切り規則なしの場合に認識される文法

```
S ::= N3 N3
N1 ::= <B> # </B>
N2 ::= N1 N1
N3 ::= <A> # N2 N2 # </A> #
```

(c) 打ち切り規則ありの場合に認識される文法

図 5　打ち切り規則追加による文法の違い
Fig. 5　Effect of adding termination rules on resulting syntax.

の対象とすることが可能となる．

繰返しでない規則については，生成規則の先頭に呼び出しもとのシンボルを追加した新たな規則を生成することでマッチングの対象とする．たとえば，**図 7** (a) において，シンボル T4 がパターンの先頭として無効なシンボルであり，T4(N2) の直前に位置するシンボル T1 および，T3 がパターンの先頭として有効なシンボルの場合，生成規則 N2 から新たな生成規則，N7，N8 を生成する．なお，変形処理においては，新しい規則の元になった規則は削

```
N0 ::= <items> N8 # </items>
N8 ::= N7*
N7 ::= # <item> # <name> # </name> … </item>
```

(a) 適用前

```
N0 ::= <items> N10 # </items>
N10 ::= # N11 <item> # <name> … </item>
N11 ::= N12*
N12 ::= <item> # <name> # </name> … </item> #
```

(b)適用後

図 6　繰返し規則の変形
Fig. 6　Transformation of repetition rules.

```
N0 ::= T1 N2 T3 N2          N0 ::= T1 N2 T3 N2
N2 ::= T4 T5 T6             N2 ::= T4 T5 T6
                            N7 ::= T1 N2
                            N8 ::= T3 N2
```

(a) 適用前　　(b) 適用後

図 7　非繰返し規則の変形
Fig. 7　Transformation of non-repetition rules.

除することはできない．これは，元規則が他の規則から呼ばれる場合には依然として必要であるためである．しかし，元規則は，後に適用される規則選択処理において無視することにより，パターンの先頭として現れないことを保証する．

3.2.4　規 則 選 択

生成された文法のすべての規則に対して最適化スキャナを生成すると，解析対象文書に対する最適化スキャナの適用率は向上する．しかし，パターン長が短いと，最適化スキャナの呼び出しオーバヘッドが隠蔽できず，かえって性能を低下させる要因になる．

そのため，学習した文法から XML 文書処理の高速化が見込める規則を選択する．選択にあたっては，(1) 規則の構造，(2) 予測される規則長，(3) 出現頻度の 3 つの要因に着目する．規則の構造では，繰返し構造の優先度が高いものとする．予測される規則長については，繰返し部分は 100 回繰り返されるものとして，規則長を求める．また，頻度は学習データに現れる回数である．規則選択処理の手順は次となる．

N11 ::= { <item> # <name> # </name> ... </item> # }*

図 8　インライン展開後の文法
Fig. 8　Rule after inlining.

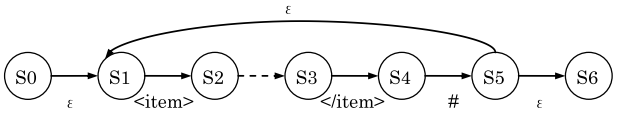


図 9　生成された NFA
Fig. 9　Generated NFA.

規則選択：

1. 次の優先度で規則を並べ替える．
   優先度：　繰返し構造 > 長さ > 頻度
2. 並べ替えた規則の先頭から次の規則に従って規則を選択する．
   ア）次の規則は除外する
   - 開始規則
   - 規則の先頭がパターンの先頭として無効なもの
   - 繰返し本体，（ピーリングの結果生成される）余り規則

   イ）選択済み規則の最大規則長の半分以下の場合は除外
   ウ）選択済み規則に包含される場合は除外

変形処理適用後の文法（図 6 (b)）に対して，上記規則選択を適用すると，N0 は開始規則，N10 はピーリングの余り規則，N12 は繰返し本体のため除外され，N11 のみが選択される．

3.2.5　認識パターン生成

選択された生成規則からマッチング用オートマトンを生成する．まず，下準備として生成した文法中の生成規則をインライン展開する．SEQUITUR によって生成した文法は再帰的な呼び出しを含まないためインライン展開は有限回で停止する．**図 8** に規則選択により選ばれている N11 に対する，インライン展開後の文法を示す．インライン展開を適用した個々の生成規則は，入力中に出現する部分トークン列に対応する生成規則を表しており，連接，繰返しからのみなる正規文法と等価である．各生成規則で表現される部分トークン列に対するマッチングを可能とするため，インライン展開を適用した生成規則からマッチング用の NFA を生成する．この NFA は，各生成規則を受理する NFA を作成し，これを開始状態および終了状態と ε 遷移によって接続することによって構成する．図 8 の文法に対して生成される NFA を**図 9** に示す．

```
if (( n = match(パターン選択テーブル) > 0 )) {
  // 最適化スキャナの呼び出し
  scanner(..., n);
} else {
  //通常のスキャナの処理
}
```

図 10　最適化スキャナ呼び出しコード
Fig. 10　Calling code skeleton of optimized scanner.

NFA は曖昧な遷移を含むため実際のマッチングに利用するには効率が悪い．そこで，NFA を DFA に変換する．この変換処理自体は標準的な DFA 構築，および，状態最小化アルゴリズムによる．

3.3　最適化スキャナ生成と呼び出し

最適化スキャナ生成処理では，DFA で表現される認識パターンから，そのパターンを読み取る最適化スキャナを生成する．解析対象の文書を読み取る際は，生成した最適化スキャナをロードし，認識パターンにマッチする場合は最適化スキャナを呼び出す．認識パターンであるか否かの判定は，DFA の初期状態からの遷移に一致するか否かの判定となる．この判定を最適化スキャナ中で行うと，パターンに一致しない場合も，最適化スキャナの呼び出しオーバヘッドがかかるため，判定処理は，最適化スキャナ呼び出し側で行う．そのため，判定に必要な初期状態からの遷移の情報をパターン判定テーブルとして作成する．パターン判定テーブルは遷移文字列と遷移先の状態番号の組のテーブルである．

図 10 は，最適化スキャナ呼び出し部分のコードである．判定処理（match）では，最適化スキャナとともにロードされた，パターン判定テーブルを利用して，次の読み取りトークンがパターンの先頭であるか否かを判定し，もし一致する遷移がある場合は，遷移先の状態番号を返す．状態番号が取得できた場合は，その番号を引数として最適化スキャナを呼び出す．状態が取得できなかった場合は通常のスキャナの処理を実行する．

3.4　最適化スキャナのコード生成

生成するスキャナコードを**図 11** に示す．DFA の 1 つの状態が，switch 文の 1 つの case ブロックに対応する．図中の「パターン判定コード」は，case ブロックが現れる状態からの遷移に対する判定となる．もし，次の入力トークンが遷移条件と一致する場合は，「アクションコード」においてその遷移に対応するイベントハンドラの呼び出しを行い，次の状態を設定する．「リカバリコード」は，パターンの途中で入力が一致しなくなった場合に呼び

出される．通常，このコードは，単純に呼び出し元へ戻るコードとなるが，属性値を持つ開始タグの読み取り途中であった場合は，そのタグの終わりまで読み取りを行い，その後，呼び出し元へ戻るコードとする．これは，最適化スキャナと通常のスキャナとの制御の移行が，タグを単位としているためである．

DFA 上で，ある状態から次の状態への遷移が分岐なく続く間は，最適化スキャナのコード上で状態を連続して並べることにより，遷移のためのコードを省略することができる．そのため，状態の配置を，DFA を初期状態から深さ優先順にたどった順序とし，2 つの状態間にお互いの状態遷移以外の遷移がない場合，遷移コードを省略する．また，ある状態への遷移がコード並び上の直前の状態のみである場合，その状態のラベルも不要のため削除する．たとえば，図 11 の場合，もし，同図 (a) の lll とその直後のラベル（同図 (c) の mmm）が同じ場合は，同図 (a)，(b) のコードは削除できる．また，ラベル mmm への遷移が他にない場合は，同図 (c) のラベルも削除可能である．

```
public class CustomScanner
  extends AbstractCustomScanner {
  public int scanner(..., int start_id) {
    int nextState = start_id;
    loop: while (true) {
      switch (nextState) {
      case nnn:                          // 状態
        if (パターン判定コード) {
          アクションコード
          nextState = lll;               (a) 遷移
          break;                         (b)
        } else {
          リカバリコード
        }
      case mmm:                          (c)
        ...
    }
  }
}
```

図 11　最適化スキャナコード
Fig. 11　Code skeleton of optimized scanner.

4. 評　　価

本章では，改良版 XML パーサの評価結果を述べる．評価は，xerces-J 2.7.1 [7] の SAX パーサに対して本稿に示す手法を実装し，本稿の手法を適用の有無での性能を比較することにより行った．また，本章に示す評価は，次の環境で行った．

CPU：Pentium4 3 GHz，Memory：1024 MB，
OS：Windows XP SP2，Java：JDK1.5.0_06
Java オプション：-server –Xms512M –Xmx512M

測定対象の XML 文書は，XML パーサのベンチマークである XMLBench [8]，XMLTest [9] に含まれる XML 文書のうち，データサイズが 100 KB 以上のファイルを対象とした．なお，測定方法，条件は次とした．

- 300 回ウォームアップ後の 300 回の処理性能を測定．
- 学習データとしては元のデータから繰返し 5 回分を手作業で切り出したデータを使用．
- 検証なし，ハンドラはすべて空．

図 12 は名前空間対応なしの場合，**図 13** は名前空間対応ありの場合の結果を，xerces-J

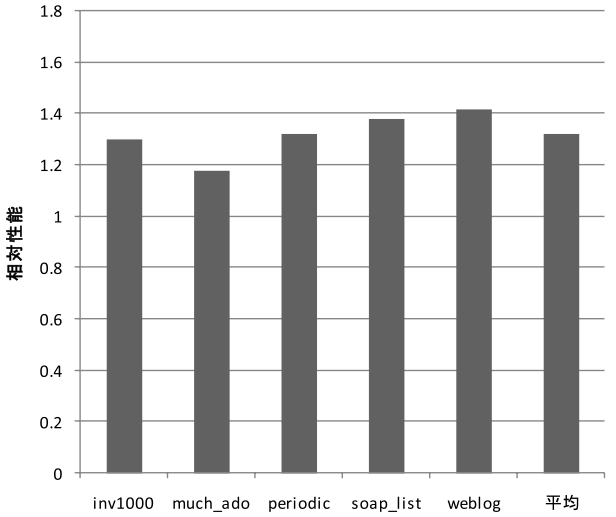


図 12　名前空間対応なしの場合の性能
Fig. 12　Performance without name space processing.

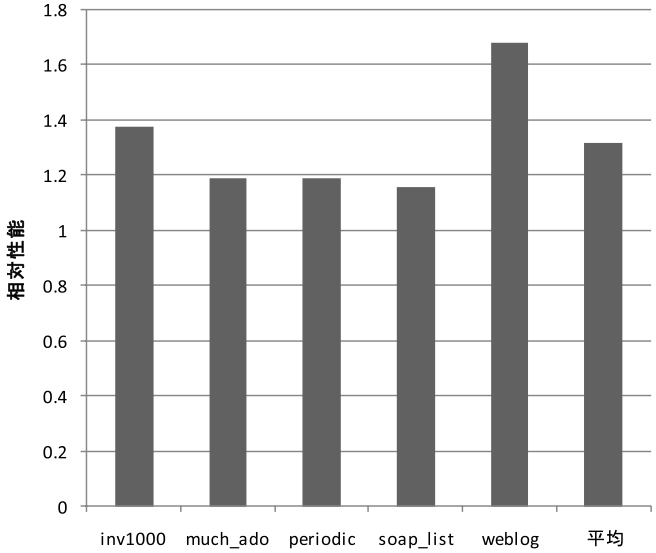


図 13　名前空間対応ありの場合の性能
Fig. 13　Performance with name space processing.

表 1　学習フェーズの結果
Table 1　Result of the learning phase.

XML 文書	生成規則数	選択規則数	DFA 状態数	最適化スキャナの Java バイトコードサイズ[byte]
inv1000	48	1	89	2220
much_ado	23	5	42	1073
periodic	24	3	167	3843
soap_list	7	1	41	912
weblog	3	1	45	821

2.7.1 の性能を 1 とした相対値で示す．図 12 より，xerces と比較して，名前空間対応なしの場合，平均 31%，最大 41%の性能向上が，名前空間対応ありの場合は，平均 31%，最大 67%の性能向上が得られていることが分かる．

学習フェーズの結果を，**表 1** に示す．生成規則数が多いことは，学習データに多様な構造が含まれていることを意味する．生成規則数は，3 から 48 まで多様であるが，選択された規則数は生成規則数によらず，いずれの場合も 5 以下と少なく抑えられていることが分かる．ただし，much_ado においては，生成規則数に対して選択された規則数の割合が高い．これは，最適化スキャナの切替え頻度が高くなる傾向にあることを意味している．これは，名前空間対応なしの場合において，much_ado の性能が低いことの一因となっている．なお，選択された規則は，periodic を除くすべての場合において最大の繰返し生成規則が選択できている．periodic においては，最大の繰返し構造が，内部に含む要素を選択可能な構成となっているため，別生成規則として学習されてしまい，1 つの繰返しパターンとして認識されていない．しかし，別々に学習された生成規則は規則選択により両者とも選択され，オートマトン上では共通部分がマージされた構成として認識されている．

5. 関 連 研 究

従来，パーサを特定の入力に対して特化することによる高速化技法に関する研究は多く行われている．ここでは，これらの研究を，特化に用いる対象が，スキーマ，実行時データ，学習データのいずれであるかで分類する．

スキーマからパーサを特化する手法として，文献 10)–12) などが提案されている．XML 文書は，その構造定義を XML Schema などのスキーマ言語によって与えることができる．文献 10)，11) の手法では，XML Schema からマッチング用の DFA を生成し，それに従って XML 文書の解析を行うことで高速化を図る．文献 12) の手法では，スキーマから直接，解析用のコードを生成する．これらの手法は，スキーマを用いて受理可能な構文を限定すること，および，スキーマで静的に決定するデータはあらかじめ作成することなどにより，高速化を図る．また，スキーマから解析器を生成することから，XML Schema の検証処理を同時に行えるため，検証時間を短縮できるメリットがある．我々の提案する手法での対象 XML 文書処理の高速化の基本的な考え方は，これらの手法と同様である．また，我々の手法では XML Schema の検証時間を短縮することはできない．しかしながら，一般的な XML 文書処理の場面では，XML Schema による構造定義が与えられているとは限らないため，これらの手法の適用対象は限定される．

構造定義が与えられない問題に関しては，これらの手法と XML 文書からスキーマを自動導出する手法（文献 13)，14) など）との組合せが考えられる．この場合，スキーマ導出の際に入力データがすべての構造を網羅できていないケースで，スキーマに一部の欠落が生じる．先に述べたスキーマからパーサを特化する手法では，スキーマを満たさない文書の読み

取りは考慮されていないため，同様に，適用対象が限定される．我々の手法では，学習データに含まれていない入力が与えられた場合においても，効率は落ちるが，読み取ることが可能である．

Web サービスのクライアント/サーバ間でやりとりされるメッセージプロトコルである SOAP は，表現形式として XML を採用している．一般に，同一 Web サービスに対するメッセージでは，類似の構造で，値のみが異なる場合が多い．そこで，このような類似性に着目した手法として，SOAP メッセージを可変部と固定部に分離し，テンプレートとして与えることによりメッセージ処理を高速化する手法が提案されている[15]．この手法は固定的な入力形式には適用可能であるが，可変データへの適用の柔軟度が低いという問題がある．

そこで，入力 XML データから動的にパターンマッチング用の DFA を構成する手法が提案されている[16),17]．これらの手法は本稿で示した解析手法と類似であるが，入力の処理と並行して DFA の構築を行う方式を採用している．そのため，これらの手法では，動的に入力データに即して DFA を更新できるという利点がある．一方，我々の手法は，学習データにより，事前に静的に DFA を構築し，その結果から最適化スキャナを生成するため，最適化スキャナ生成時において，インライン展開を適用することにより，トークン読み取り処理を，パターンに特化した構成にでき，また，規則選択により，処理の高速化が想定できる規則を選択することが可能である．なお，我々の手法においても，実行時データから学習する構成とすることも可能である．しかしながら，文法圧縮処理などのオーバヘッドのため，有用な結果を得ることは容易でないと想定される．

本稿の提案手法の特徴として学習データから頻出パターンを構成しパーサを特化する点があげられる．このように，学習データを用いてパーサを特化する手法は我々の知る限り見つかっていない．また，文法変換による圧縮手法を用いた XML 文書からの頻出パターンの認識，および，その XML パーサの高速化への適用例も，我々の知る限り見つかっていない．

6. 結　　論

本研究では，学習したパターンに対して予測的に構文解析することにより，処理時間の短縮を図る改良版 XML パーサの開発を行った．改良版 XML パーサでは，あらかじめ，学習データから頻出文法を学習し，その文法を読み取る最適化スキャナを生成する．実行時は，学習データと一致する部分は最適化スキャナを使用することで，予測的に構文解析を行う．

本手法を適用することにより，解析対象の XML 文書と類似の構造を持つ XML 文書を学習文書として与えた場合，名前空間対応ありの場合，SAX 処理性能が平均で 31%，最大で 67%向上することを確認した．

XML スキーマが与えられるようなケースでは，5 章で示したスキーマベースの手法がより良い性能を得る可能性がある．今後の課題として，これら関連研究に対する定量的な評価があげられる．

改良版 XML パーサの課題として，繰返し中の要素の出現が選択可能なパターンの学習への対応があげられる．評価で述べたように，periodic に対しては，文法生成時に最適なパターンを学習できていない．そのため，後の繰返し認識部において，繰返しを認識できず，繰返しごとに最適化スキャナが呼び出される．もし，文法生成時に繰返しを認識できた場合，呼び出しオーバヘッドを削減できる．繰返しを認識できたものと仮定して実験を行った結果，SAX の場合の性能が約 10%向上することを確認している．

参 考 文 献

1) Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E. and Yergeau, F.: Extensible Markup Language (XML) 1.0, 3rd Edition (2004). http://www.w3.org/TR/REC-xml
2) W3C: SOAP Specifications (2004). http://www.w3.org/TR/soap
3) W3C: Document Object Model (DOM) Technical Reports (2004). http://www.w3.org/DOM/DOMTR
4) Meggison, D.: SAX: Simple API for XML (2004). http://www.saxproject.org
5) Kieffer, J. and Yang, E.: Grammer-Based Codes: A New Class of Universal Lossless Source Codes, *IEEE Trans. on Information Theory*, Vol.46, No.4, pp.737–754 (2000).
6) Nevill-Manning, C.G. and Witten, I.H.: Identifying Hierarchical Structure in Sequences: A linear-time algorithm, *Journal of Artificial Intelligence Research*, pp.67–82 (1997).
7) Xerces. http://xerces.apache.org
8) Sosnoski Software Solutions: XMLBench Document Model Benchmark (2006). http://www.sosnoski.com/opensrc/xmlbench
9) Sun Microsystems Inc.: XMLTest. http://java.sun.com/performance/reference/codesamples
10) Chiu, K. and Lu, W.: A Compiler-Based Approach to Schema-Specific XML Parsing, *1st International Workshop on High Performance XML Processing* (2004).
11) Engelen, R.: Constructing Finite State Automata for High Performance XML WebServices, *Proc. International Symposium on Web Services*, pp.975–981 (2004).
12) Perkins, E., Matsa, M., Kostoulas, M.G., Heifets, A. and Mendelsohn, N.: Gener-

ation of Efficient Parsers Through Direct Compilation of XML Schema Grammars, *IBM Systems Journal*, Vol.45, No.2, pp.225–244 (2006).
13) Clark, J.: Trang Multi-format schema converter based on RELAX NG (2003). http://www.thaiopensource.com/relaxng/trang.html
14) Hegewald, J., Naumann, F. and Weis, M.: XStruct: Efficient Schema Extraction from Multiple and Large XML Documents, *Proc. 22nd International Conference on Data Engineering Workshops*, p.81 (2006).
15) Takeuchi, Y., Okamoto, T., Yokoyama, K. and Matsuda, S.: A Differential-analysis Approach for Improving SOAP Processing Performance, *Proc. 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service*, pp.472–479 (2005).
16) Abu-Ghazaleh, N. and Lewis, M.J.: Differential Deserialization for Optimized SOAP Performance, *Proc. 2005 ACM/IEEE Conference on Supercomputing*, pp.21–31 (2005).
17) Takase, T., Miyashita, H., Suzumura, T., Takase, M. and Tatsubori, M.: An Adaptive, Fast and Safe XML Parser Based on Byte Sequence Memorization, *Proc. 14th International World Wide Web Conference*, pp.692–701 (2005).

(平成 19 年 10 月 12 日受付)
(平成 20 年 4 月 8 日採録)

太田　智也（正会員）

1998 年静岡大学大学院電子科学研究科博士課程修了．博士（工学）．1998 年より（株）日立製作所システム開発研究所勤務．言語処理系の研究開発に従事．

西山　博泰（正会員）

1993 年筑波大学大学院工学研究科博士課程修了．（株）日立製作所システム開発研究所主任研究員．最適化コンパイラ，Java 実行環境等の言語処理系の研究に従事．ACM，IEEE 各会員．

田村　清朗

1994 年東北大学大学院情報科学研究科修士課程修了．1994 年より（株）日立製作所ソフトウェア事業部勤務．COBOL コンパイラ，XML パーサの製品開発に従事．

宗近　秀生

2005 年明治大学理工学部電子通信工学科卒業．2005 年より（株）日立製作所ソフトウェア事業部に勤務．XML パーサの製品開発に従事．