# Shape Expression Schemaにおける Property Pathを用いた問合せの充足可能性判定

前田　祐紀†　　鈴木　伸崇††

† 筑波大学情報学群知識情報・図書館学類　〒 305–8550 茨城県つくば市春日 1–2
†† 筑波大学図書館情報メディア系　〒 305–8550 茨城県つくば市春日 1–2
E-mail: †s1611551@s.tsukuba.ac.jp, ††nsuzuki@slis.tsukuba.ac.jp

**あらまし**　Shape Expression Schema (ShEx) はグラフデータの構造をより適切に示すため，新たに提案されたスキーマ言語である．一般的に，グラフデータのサイズは非常に大きく，そのようなグラフデータに対して充足不能な問合せを行うのは非効率的である．そこで，与えられた問合せ式が充足不能であるかどうかを ShEx の定義から判定する手法を考える．本研究では問合せ方法を SPARQL 1.1 で定義されている Property Path とし，オートマトンを利用する手法でアルゴリズムを考案した．この考案アルゴリズムによって問合せの実行時間を短縮できることが確認された．
**キーワード**　Shape Expression Schema，Property Path，充足可能性判定

## 1 はじめに

近年，RDF データ (グラフデータ) の社会への普及が進んでいる．しかし，グラフデータの構造を示すためのスキーマの活用は未だ途上の域にある．グラフデータに対するスキーマとしては，これまでグラフスキーマなどが提案されているが，いずれも繰り返しや選言などの指定ができず，表現力が不十分である．また，RDF データのスキーマ言語として RDF Schema (RDFS) [1] が提案されているが，RDFS はオントロジー定義言語としての性質が強く，スキーマ定義には必ずしも適していない．そこで，従来のスキーマよりも適切にグラフデータの構造を記述することが可能な Shape Expression Schema (以下 ShEx) [2] が提案されている．ShEx はノードに対する型を Regular Bag Expression という規則に基づいて定義することが特徴のスキーマ言語である．

本研究では，ShEx における問合せ式の充足可能性判定について考える．問合せ式としては，SPARQL 1.1 で定義されている Property Path [3] を対象とする．ここで，問合せ式 $P$ と ShEx $S$ に対して，$S$ に妥当などのグラフデータにおいても $P$ の解が空である場合，$P$ は充足不能であるという．問合せ式 $P$ が与えられた場合，$P$ を用いてグラフデータを探索する必要がある．しかし，一般的にグラフデータのサイズは非常に大きい．したがって，$P$ が充足不能な場合，莫大な量のデータに対してそのような探索を行うのは明らかに非効率的である．一方，ShEx はグラフデータよりサイズが著しく小さく，充足可能性判定は効率よく行えると期待される．問合せ式の実行に先立ってその充足可能性判定を行うことにより，充足不能な問合せ式の実行を回避することができる．

先行研究として，ShEx に対してパターン問合せを用いての充足可能性判定を行うアルゴリズムの提案がある [4]．パターン問合せは探索したいグラフの形 (パターン) を指定してそのパターンにマッチする部分をグラフデータから探索する問合せ方法である．本研究の対象である Property Path では，探索したい経路に対する条件を記述し，指定した経路の開始ノードと終端ノードのペアが解となる．例として,「あるラベルが 0 回以上出現する」という条件に対しては指定のラベルが何回出現しても，もしくは出現しなくてもマッチするということになる．このように，両者は互いに異なる問合せであり，充足可能性の判定には異なる手法が必要となる．

本研究では，グラフデータに比べてサイズの小さい ShEx から Property Path を用いた問合せの充足可能性を判定するアルゴリズムを提案する．具体的には，ShEx と Property Path をそれぞれオートマトン $M_S, M_P$ に変換する．オートマトンとは状態と遷移の集合によって自動機械の仕組みを表すモデルである．そして，$M_S$ の各ノードを $M_P$ の開始状態と見立てて $M_P$ の受理状態まで遷移できるかを調べる．一つでも条件を満たすパスが見つかれば充足可能，見つからなければ充足不能として処理を終了する．ShEx に対して Property Path を用いた問合せの充足可能性問題を解くアルゴリズムは著者の知る限り提案されていない．

以上のアルゴリズムを実装し，充足不能な Property Path を用いて評価実験を行なった．その結果，提案アルゴリズムで充足可能性判定を行うことができ，充足可能性判定に要する時間はグラフデータの問合せ時間と比較して大幅に小さいことを確認した．

本稿の構成は以下の通りである．第 2 章では，グラフ，ShEx，Property Path，充足可能性問題の定義を述べる．第 3 章では，提案アルゴリズムについて述べる．第 4 章では，評価実験について述べる．第 5 章では，本研究のまとめを述べる．

## 2 諸定義

本章では，本研究で扱うグラフ，Shape Expression Schema，

Property Path，充足可能性問題について述べる．

### 2.1 グラフ

本研究におけるグラフはラベル付き有向グラフを用いる．$\Sigma$ 上のグラフ $G$ を，$G = (V, E)$ の 2 次組として定義する．ここで，$\Sigma$はラベルの集合，$V$ はノードの集合，$E \subseteq V \times \Sigma \times V$ はラベル付き有向辺の集合である．例として，図 1 にグラフ $G_0$ を示す．

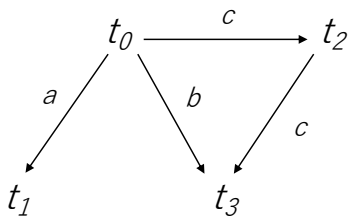


図 1　グラフ $G_0$

定義に基づいて図 1 のグラフ $G_0 = (V, E)$ を表記すると，

$$\Sigma = \{a, b, c\}$$
$$V = \{t_0, t_1, t_2, t_3\}$$
$$E = \{(t_0, a, t_1), (t_0, b, t_3), (t_0, c, t_2), (t_2, c, t_3)\}$$

となる．また，グラフ内のある辺において，辺を出力するノードを出力ノード，受け取るノードを入力ノードと呼称する．

### 2.2 Shape Expression Schema

ShEx はグラフデータに対するスキーマ言語の一つである．スキーマ言語とは XML における DTD や XML Schema のように，グラフデータの構造を定義するための言語である．ShEx では，ノードに対する型の定義について Regular Bag Expression (以下 RBE) [5] という規則を用いている．RBE は '||' を用いた連結の際に順序が無視されるという特性を持つ．RBE の規則は以下のようになっている．

- $\varepsilon$ と任意の $a \in \Sigma$ は RBE である．
- $r_1, \cdots, r_k$ を RBE とするとき，$r_1||r_2||\cdots||r_k$ は RBE である．ここで，'||' は順序を無視した連結を表す．
- $r_1, \cdots, r_k$ を RBE とするとき，$r_1|r_2|\cdots|r_k$ は RBE である．ここで，'|' は選言を表す．
- $r$ を RBE とするとき，$r?$, $r+$, $r^*$ は RBE である．

ShEx は $S = (\Sigma, \Gamma, \delta)$ と定義される．ここで，$\Sigma$ はラベルの集合，$\Gamma$ は型名の集合，$\delta$ は RBE に基づいた型を定義する関数である．以下は ShEx の一例である．

$S$ :

$$\Sigma = \{a, b, c\}$$
$$\Gamma = \{t_0, t_1, t_2, t_3, t_4\}$$
$$\delta(t_0) = (a :: t_1 | b :: t_3 | c :: t_2)^*$$
$$\delta(t_1) = b :: t_3 | c :: t_4$$
$$\delta(t_2) = c :: t_3$$
$$\delta(t_3) = \varepsilon$$
$$\delta(t_4) = a :: t_3$$

ShEx には，1 つのノードが 1 つの型を持つ single type semantics と複数の型を持つ multi type semantics が存在するが，本研究では single type semantics のみを対象とする．

### 2.3 Property Path

Property Path は RDF データに対する問合せの一種である．Property Path の仕様は SPARQL 1.1 の仕様の中に含まれており，XML の XPath のように探索したい経路に対する条件を記述する．

Property Path は正規表現とほぼ同様の定義がされる．ただし，いくつかの機能によって拡張されている．$\Sigma$ をラベルの集合としたとき，$\Sigma$ 上での Property Path の定義は以下のようになっている．

- $\varepsilon$ および任意の $a \in \Sigma$ は Property Path である．
- *はワイルドカードを表す Property Path である．
- $\{a_1, \cdots, a_k\}$ をラベル集合としたとき，$!\{a_1, \cdots, a_k\}$ は $a_1, \cdots, a_k$ 以外のラベルにマッチする Property Path である．
- 任意の a $\in\Sigma$に対して，$a^{-1}$ はラベル a のエッジを逆向きに辿る Property Path である．
- $r_1, \cdots, r_k$ を Property Path とするとき，$r_1.r_2.\cdots.r_k$ は Property Path である．ここで，'.' は連結を表す．
- $r_1, \cdots, r_k$ を Property Path とするとき，$r_1|r_2|\cdots|r_k$ は Property Path である．ここで，'|' は選言を表す．
- $r$ を Property Path とするとき，$r?$, $r+$, $r^*$ は Property Path である．

例えば，$a.c.a$ や $b.a+$ は 2.2 節の ShEx $S$ 上の PropertyPath である．

### 2.4 充足可能性問題

本研究では，問合せとして Property Path を用いている．Property Path $P$ に対して，グラフ上の経路でそのラベル列が $P$ を満たすものであれば，その経路の開始ノードと終端ノードのペアが $P$ の解となる．Property Path $P$ と ShEx $S$ に対して，「$S$ に妥当なあるグラフデータに対して空でない $P$ の解が存在する」ときに $P$ は $S$ の下で充足可能であるという．逆に，$S$ に妥当などのグラフデータに対しても $S$ の解が空となるとき，$P$ は $S$ の下で充足不能であるという．このとき，問合せが充足可能であるかどうかを判定する問題を充足可能性問題という．

例えば，2.2 節の ShEx $S$ において 2.3 節で挙げた Property Path を考えると，$a.c.a$ は $S$ の下で充足可能であり，一方 $b.a+$ は $S$ の下で充足不能である．

## 3 提案手法

本章では，ShEx における Property Path を用いた問合せの充足可能性判定アルゴリズムについて述べる．

### 3.1 アルゴリズム

本節では，ShEx における Property Path を用いた問合せの充足可能性判定アルゴリズムに関して述べる．

提案アルゴリズムでは，オートマトンを利用する．オートマ

トンは，状態とその遷移関数の集合によって自動機械の仕組みを表すモデルである．その定義を以下に示す．

- オートマトン $M$ は $M = (Q, \Sigma, \delta, q_I, F)$ によって表される．
- $Q$：状態の集合
- $\Sigma$：ラベルの集合
- $\delta : Q \times \Sigma \to Q$ (遷移関数)
- $q_I$：開始状態
- $F$：受理状態

まず，ShEx $S$ をオートマトン $M_S$ に変換する．ここで，スキーマは開始状態と受理状態という概念を持たないため，$M_S$ には $q_I$, $F$ が存在しない．ShEx の定義に従って変換を行うと以下のように表せる．

- ShEx $S = (\Sigma_1, \Gamma_1, \delta_1)$ からオートマトン $M_S = (Q_2, \Sigma_2, \delta_2, nil, \emptyset)$ に変換する
- $Q_2 = \Gamma_1$
- $\Sigma_2 = \Sigma_1$
- $\delta_2(t, a) = \{t' \mid a :: t'$ が $\delta_1(t)$ に出現する $\}$

本研究は Property Path を用いた問合せの充足可能性判定が目的であるため，型の定義におけるエッジの出現回数，及び選言は考慮する必要がない．すなわち，0 回または 1 回の出現を表す '?'，1 回以上の出現を表す '+'，0 回以上の出現を表す '*' はそれぞれ 1 回の出現とする．さらに，型の定義内で選言を含むノードを一度通ったとしても再び到達するのは同じ型の異なるノードとみなせるため，選言を表す '|' は順序を無視した連結を表現する '||' と同じとみなしてよい．

次に，Property Path で記述されたクエリをオートマトン $M_P$ に変換する．Property Path は正規表現を含んでいるが，正規表現からオートマトンへの変換方法は既に存在しており [6]，これを利用している．

ここまでの処理ののち，$M_S$ が $M_P$ を含んでいるかを調べる．提案アルゴリズムを Algorithm 1 に示す．また，本アルゴリズムでは Property Path において 1 つのラベルに!と-1 が同時に出現することはないと仮定している．

**Algorithm 1** PropertyPath Satisfiability

**Input:** ShEx schema $S = (\Sigma, \Gamma, \delta)$, Property Path $P$
**Output:** 充足可能かどうか

```
1: M_S :=ShExToAutomaton(S);
2: M_P :=PpToAutomaton(P);
3: Σ' := Σ(M_S) ∪ Σ(M_P);
4: for each n ∈ Q(M_S) do
5:     curstate := [0, n];
6:     arrives ← curstate;
7:     TransCheck(M_S, M_P, curstate, arrives, Σ');
8: end for
9: report 充足不能;
```

**Algorithm 2** TransCheck

**Input:** $M_S, M_P, curstate, arrives, \Sigma'$

```
1: for each sn ∈ Σ(M_P) do
2:     if sn[0] = '!' then
3:         Excl =MakeExclamation(M_P, curstate);
4:         for each sm ∈ Σ' do
5:             if sm ∉ Excl then
6:                 ExclTransNode(M_S, M_P, sm, sn, curstate, arrives, Σ');
7:             end if
8:         end for
9:     else
10:        if sn[−1] = '−' then
11:            MinusTransNode(M_S, M_P, sn, curstate, arrives, Σ');
12:        else
13:            TransNode(M_S, M_P, sn, curstate, arrives, Σ');
14:        end if
15:    end if
16: end for
```

**Algorithm 3** TransNode

**Input:** $M_S, M_P, sn, curstate, arrives, \Sigma'$

```
1: ppstate = δ(M_P)(curstate[0], sn)
2: for each shexstate ∈ δ(M_S)(curstate[1], sn) do
3:     curstate = [ppstate, shexstate]
4:     ArriveCheck(M_S, M_P, curstate, arrives, Σ')
5: end for
```

**Algorithm 4** ArriveCheck

**Input:** $M_S, M_P, curstate, arrives, \Sigma'$

```
1: if ∀s ∈ curstate (s ≠ nil ∧ s ≧ 0) then
2:     if curstate[0] ∈ F(M_P) then
3:         report 充足可能;
4:     else
5:         if curstate ∉ arrives then
6:             arrives ← curstate;
7:             TransCheck(M_S, M_P, curstate, arrives, Σ');
8:         end if
9:     end if
10: end if
```

まず，1〜2 行目で ShEx で記述されたグラフスキーマと Property Path で記述されたクエリをオートマトン $M_S, M_P$ に変換する．ここで変換のために呼び出す関数について説明する．

- ShExToAutomaton は ShEx をオートマトンに変換する関数である．本節冒頭で述べた通り，グラフスキーマはオートマトンの定義のうち開始状態と受理状態の概念を持たず，残る集合は ShEx の定義から変換可能である．
- PpToAutomaton は Property Path をオートマトンに変換する関数である．前章の 2.3 節にある通り Property Path は正規表現とほぼ同様の定義がされるがいくつかの機能によって拡張されている．正規表現をオートマトンに変換するアルゴリズムは既に考案されており，正規表現部分についてはこれを

利用して変換を行なっている．また，正規表現にはない特殊記号についてもこの関数の中で処理を行なっている．-1 は該当するラベルに「-」をつけ，!は該当するラベルに「!」をつける．例えば，$a^{-1}$ というラベルは $a-$ に，$!\{b\}$ というラベルは $!b$ に変換している．つまり，Property Path のオートマトンにおいて，-1 に関してはラベル $a$ のエッジを逆向きに辿るのではなく，「ラベル $a-$ で伸びるエッジを辿る」とする．同様にして，!に関しては b 以外のラベルで辿るのではなく，「ラベル $!b$ で伸びるエッジを辿る」とする．ここでラベルに付与した「-」や「!」に応じて充足可能性判定の処理を行う．

以下，$M_S$ のラベル集合，状態集合，遷移関数をそれぞれ $\Sigma(M_S)$，$Q(M_S)$，$\delta(M_S)$ と表す．$M_P$ も同様の表記を用いる．

3 行目で $M_S, M_P$ のラベルをまとめた集合 $\Sigma'$ を用意する．4 行目からは $M_S$ の各ノード $n$ に対して $n$ を Property Path の開始状態と見立てて TransCheck 関数を再帰的に呼び出す．*curstate* は現在の状態を示しており，より具体的には $M_P$ における現在の状態と $M_S$ における状態の組である．*arrives* は既に出現したノード組を格納するための配列であり，*curstate* をループ開始時に格納している．

Algorithm 2 に示した TransCheck 関数では $M_P$ の各ラベル *sn* で現在いるノードからの遷移を調べる．*sn* の値に対して以下のように処理を行う．

- *sn* に「!」も「-」も含まれていない場合は TransNode 関数を呼び出して *curstate* を遷移させる．TransNode 関数は Algorithm 3 に示している．これは，指定したラベル *sn* で現在いるノードから遷移させる関数である．遷移後に ArriveCheck 関数を呼び出している．
- *sn* が「!」を含む場合，MakeExclamation 関数を用いて調べる対象から外すラベル集合 *Excl* を求める．MakeExclamation は $M_P$ の現在のノードから伸びるエッジのうち「!」付きのラベル集合を求める関数である．例えば，現在のノードからラベル「$!\{a|b\}$」が伸びている Property Path の場合には $a, b$ が格納される．これは例で挙げたように「!」で指定されるラベルが 1 つではなく「|」を用いて複数存在する場合があり，それらは全て選択しないようにする必要があるためである．次に $\Sigma'$ の各ラベル *sm* について，*Excl* に含まれていない場合に ExclTransNode 関数を呼び出して *curstate* を遷移させる．その際，$M_S$ の遷移ラベルは *sm*，$M_P$ の遷移ラベルは *sn* となる．遷移後は TransNode 関数と同様に ArriveCheck 関数を呼び出している．
- *sn* が「-」を含む場合，MinusTransNode 関数を呼び出して *curstate* を遷移させる．MinusTransNode は指定したラベル *sn* で $\Gamma(M_S)$ の各ノードから遷移できるかを調べ，遷移できる場合はそのノードを $M_S$ の遷移後の位置として遷移させる関数である．遷移後は TransNode 関数と同様に ArriveCheck 関数を呼び出している．

それぞれの関数を用いて *curstate* を遷移させた後はいずれも ArriveCheck 関数を呼び出している．ArriveCheck を Algorithm 4 に示す．これは与えられた Property Path が充足可能かの判定と $M_S, M_P$ のノード組が既に到達したものであるかを判定する関数である．まず，*curstate* の各要素 $s$，すなわち $M_S, M_P$ の現在いるノードが有効なノードであるかを確認する．変換後のオートマトンはどちらもノードを 0 以上の整数で表し，遷移できないラベルでの遷移先には-1 を格納している．つまり，全ての $s$ が null でない，かつ負でない時に *curstate* が有効として処理を進める．その後，現在の $M_P$ のノードが $M_P$ の受理状態の集合に含まれていれば充足可能と出力して処理を終了する．含まれていない場合のうち，*curstate* が *arrives* に含まれていなければ *curstate* を *arrives* に格納し，再度 TransCheck 関数を呼び出す．*curstate* が一度到達したノード組である場合，そこから遷移できるノードは一度目に到達した時と全く同じになるので遷移処理を進める必要がないためである．

上記の処理を繰り返していき，いずれの遷移ルートも条件を満たさない場合には充足不能と出力する．

### 3.2 アルゴリズムの動作例

本節では，TransCheck の具体的な動きを説明する．以下の Property Path と ShEx を考える．

| Property Path | ShEx |
|---|---|
| $(b\|(c.a)).c^{-1}.b$ | $\Sigma = \{a, b, c\}$<br>$\Gamma = \{t_0, t_1, t_2, t_3\}$<br>$\delta(t_0) = (a :: t_1 \| c :: t_2)*$<br>$\delta(t_1) = b :: t_3 \| c :: t_4$<br>$\delta(t_2) = c :: t_3$<br>$\delta(t_3) = b :: t_0$<br>$\delta(t_4) = a :: t_3$ |

上記の ShEx と Property Path から ShExToAutomaton と PpToAutomaton の処理を通して図 2 のようなオートマトンを作成する．これらをそれぞれ $M_S, M_P$ とする．$M_S, M_P$ は具体的には以下の情報が格納されている．ここでは遷移先がないラベルでの遷移関数は表記していないが，実際には「-1」に遷移するという形で格納されている．

| Property Path | ShEx |
|---|---|
| $Q = \{u_0, u_1, u_2, u_3, u_4\}$ | $Q = \{t_0, t_1, t_2, t_3\}$ |
| $\Sigma = \{a, b, c, c-\}$ | $\Sigma = \{a, b, c\}$ |
| $\delta = (u_0, c, u_2), (u_0, b, u_1), (u_2, a, u_1),$<br>$(u_1, c-, u_3), (u_3, b, u_4)$ | $\delta = (t_0, a, t_1), (t_0, c, t_2),$<br>$(t_1, b, t_3), (t_1, c, t_4),$<br>$(t_2, c, t_3), (t_3, b, t_0),$<br>$(t_4, a, t_3)$ |
| $q_I = \{u_0\}$ | |
| $F = \{u_4\}$ | |

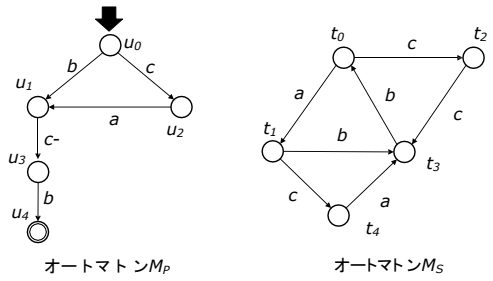


図 2　マッチング

最初に，*curstate* = $[u_0, t_0]$ として処理を開始する．この時に *arrives* に $[u_0, t_0]$ が格納される．図 3 は *curstate* の位置に印をつけたものである．ラベルを $b$ として遷移させると，*curstate* = $[u_1, -1]$ となる．この *curstate* には遷移できていないことを示す-1 が格納されているため，ArriveCheck が呼び出された際に無効と判定され処理が止まる．次に，ラベルを $c$ として遷移させると，*curstate* = $[u_2, t_2]$ となる．この *curstate* は正常に遷移できているため ArriveCheck で有効と判定される．$u_2$ は $M_P$ の受理状態に含まれていないため，充足可能と判定されず処理が続けられる．また，$[u_2, t_2]$ は *arrives* にも含まれていないため *arrives* に格納し，これを *curstate* として再度 TransCheck を呼び出す．

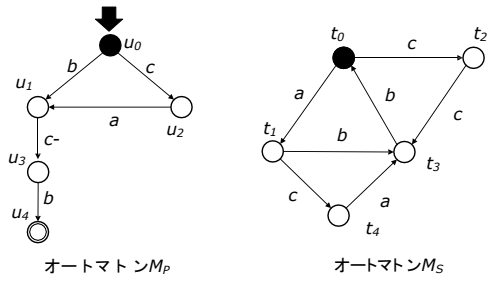


図 3　マッチング 1-1

図 4 は *curstate* = $[u_2, t_2]$ の状態の図である．前述の手順と同様に進めるが，$u_2$ から遷移できるラベルは $a$ のみなのに対して $t_2$ は $a$ では遷移できないため，ここで処理が止まる．これで開始状態を $t_0$ とした時の遷移ルートは全て調べたので，別のノードを開始状態として同様の処理を行なっていく．

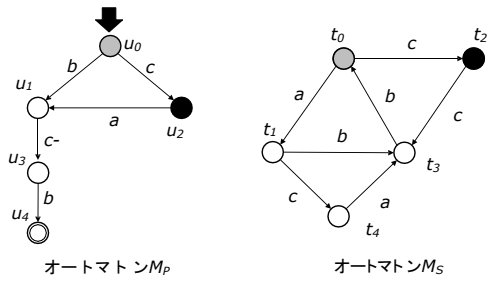


図 4　マッチング 1-2

次に，*curstate* = $[u_0, t_1]$ として処理を開始する．*arrives* は開始状態のノードが切り替わる際に初期化されるため，先程到達したノード組は含まれていない．図 5 は *curstate* を切り替えて *arrives* を初期化した後の図である．ラベルを $b$ として遷移させると *curstate* = $[u_1, t_3]$ となり，$u_1$ が受理状態でなく *curstate* が *arrives* にも含まれていないため *arrives* に追加したのち *curstate* = $[u_1, t_3]$ として TransCheck を呼び出す．

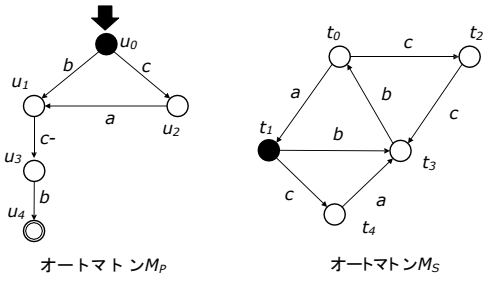


図 5　マッチング 2-1

図 6 は *curstate* = $[u_1, t_3]$ の状態を表した図である．次はラベル $c-$ のため MinusTransNode を呼び出す．具体的には $M_S$ の各ノードからラベル $c$ で $t_3$ に遷移できるノードがあるかを調べている．ここでは $t_2$ が条件を満たすため，遷移後は *curstate* = $[u_3, t_2]$ として ArriveCheck を呼び出す．この例では遷移先は 1 つだが，該当するノードが複数ある場合も全ての *curstate* で ArriveCheck を呼び出す．$u_3$ が受理状態に含まれておらず，$[u_3, t_2]$ も *arrives* に含まれていないため，ノード組を格納して TransCheck を呼び出す．

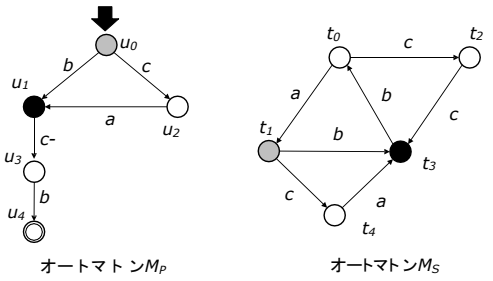


図 6　マッチング 2-2

図 7 は *curstate* = $[u_3, t_2]$ の状態であるが，$u_3$ から遷移するためのラベル $b$ では $t_3$ から遷移できないためここで処理が止まる．

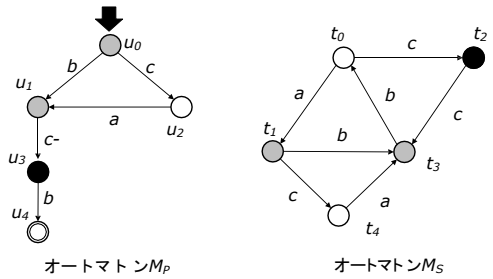


図 7　マッチング 2-3

次は $curstate = [u_0, t_1]$ まで戻り，ラベル $c$ での遷移を調べていく処理を行う．図 8 は到達済みのノード組に印をつけたまま図 5 の $curstate$ まで戻った状態を表している．同様の処理を施すと，ラベル $c$ で $[u_2, t_4]$，その後ラベル $a$ で $[u_1, t_3]$ まで遷移することができる．ここで，$[u_1, t_3]$ は $arrives$ に格納されている．すなわち，既に到達したノード組であり，ここから遷移できる可能性のあるルートは全て探索済みであるため，この時点で処理を終了する．これで $t_1$ を開始状態とした時の遷移ルートは全て調べたと言える．

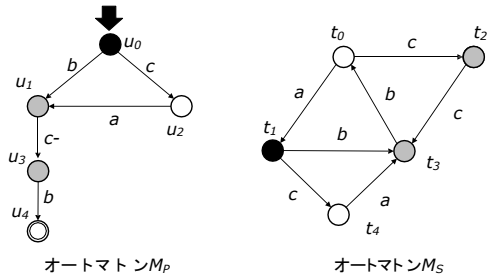


図 8　マッチング 3-1

同様にして，$t_2, t_3, t_4$ を開始状態として処理を実行する．ここでは省略するが，いずれの場合も条件を満たす遷移ルートはなく，$M_S$ のどのノードを開始状態に見立てた場合においても条件を満たさないため充足不能と出力し終了する．

## 4　評 価 実 験

本章では，前章での提案アルゴリズムに関する評価実験について述べる．

### 4.1　概　　要

まず，前章で述べた提案アルゴリズムを Ruby を用いて実装した．次に，SP$^2$Bench と BSBM から作成したグラフデータ，及び二者に妥当な ShEx をそれぞれ用意した．そして，グラフデータと ShEx に対して充足不能な Property Path を作成し，両者の問合せに要した時間を計測した．

SP$^2$Bench [7] は，コンピュータ科学に関する書誌学ウェブサイトの DBLP に基づき任意のトリプル数もしくはデータサイズの RDF データを生成する SPARQL ベンチマークソフトである．図 9 は SP$^2$Bench のデータ構造，図 10 は DBLP と SP$^2$Bench のラベルの対応関係を表した図である (いずれも文献 [7] から引用している)．生成されるデータは，各行ごとに名前空間もしくはトリプルを表している．名前空間には DBLP 固有の文書クラスを定義している．トリプルは (出力ノード ラベル 入力ノード .) といったように要素を空白で区切って記述している．

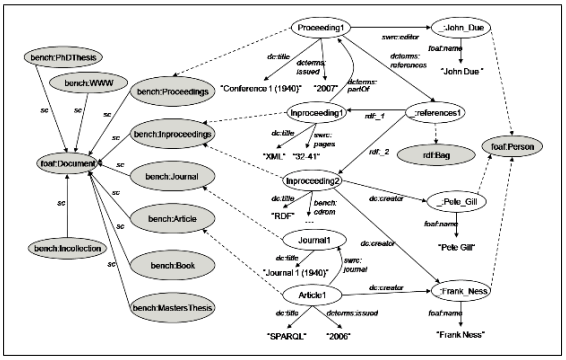


図 9　SP$^2$Bench のデータ構造

BSBM [8] は製品の電子商取引を使用事例として想定し，任意のデータサイズの RDF データを生成する SPARQL ベンチマークソフトである．図 11 は文献 [8] から引用した BSBM のデータ構造を表した図である．生成されるデータは，各行ごとにトリプルを表している．

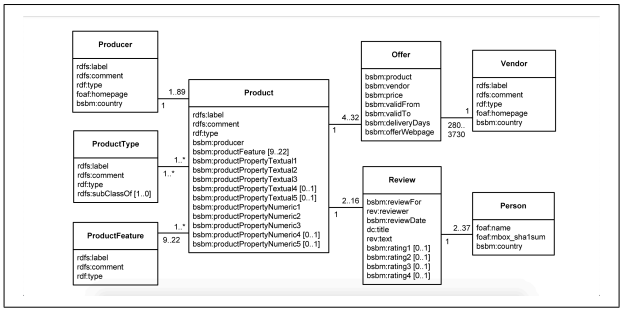


図 10　BSBM のデータ構造

評価実験の環境は以下の通りである．

**プロセッサ**　2.3 GHz デュアルコア Intel Core i5
**主記憶**　8.00 GB (4.04 GB 使用可能)
**OS**　macOS Catalina 10.15.1
**使用言語**　ruby 2.6.3

Property Path の検索をグラフデータに対して行う際は，Ruby の sparql ライブラリを使用した．sparql ライブラリは Ruby 上で SPARQL 1.1 の検索を行えるようにするライブラリである．ただし，SP$^2$Bench から生成されるグラフデータは Notation3 というフォーマットを使用しているが，sparql ライブラリではこのフォーマットが対応していないため，対応しているフォーマットの一つである N-Triple へ変換している．変換には RDF Translator [9] というツールを利用した．本研究の評価実験では，SP$^2$Bench で生成したトリプル数 10000 のデータ

を N-Triple に変換したデータを使用する．変換後のデータはトリプル数 10000，サイズが 1,715,705B である．BSBM で生成したデータはトリプル数 40377，サイズが 10,216,303B である．

評価実験にあたってそれぞれの ShEx に対して充足不能な Property Path を 4 種類ずつ作成した．SP$^2$Bench に対する Property Path を PP1, PP2, PP3, PP4 とし，BSBM に対する Property Path を PP5, PP6, PP7, PP8 とする．これらはそれぞれ後半の Property Path ほど長さが長くなる，もしくは「?」等の記号が含まれて複雑になるように作成した．

### 4.2 結　　果

前節で述べた ShEx と Property Path を用いて提案手法により充足不能判定を行い，さらに前節で述べた RDF データと Property Path を用いて問合せ処理を実行した．SP$^2$Bench における両者の実行時間および時間比率を表 4.1 に示す．また，BSBM における両者の実行時間および時間比率を表 4.2 に示す．計測には Ruby の Benchmark モジュールを用いており，単位は秒である．

表 1　SP$^2$Bench における実行結果

| | PP1 | PP2 | PP3 | PP4 | 平均 |
|---|---|---|---|---|---|
| 充足不能性判定 | 0.0069 | 0.0063 | 0.0065 | 0.0064 | 0.0065 |
| RDF データ問合せ | 4.4870 | 8.2573 | 28.802 | 62.104 | 25.913 |
| 時間比率 | 0.0015 | 0.0008 | 0.0002 | 0.0001 | 0.0007 |

表 2　BSBM における実行結果

| | PP5 | PP6 | PP7 | PP8 | 平均 |
|---|---|---|---|---|---|
| 充足不能性判定 | 0.0063 | 0.0065 | 0.0066 | 0.0099 | 0.0073 |
| RDF データ問合せ | 19.667 | 26.902 | 69.969 | 298.82 | 103.84 |
| 時間比率 | 0.0003 | 0.0002 | 0.00009 | 0.00003 | 0.00016 |

上記の結果を見ると，提案アルゴリズムによって ShEx から探索を行うと，Property Path が充足不能であった場合，判定に要する時間はグラフデータでの探索に対して大幅に短縮できることが分かる．また，Property Path が長くなる場合や複雑な形になる場合においても，グラフデータでの探索は大幅に実行時間が増加するのに対して ShEx での探索は実行時間の増加を抑えられており，より顕著に差が現れている．

## 5 む　す　び

本稿では，ShEx のスキーマと Property Path で記述されたクエリをオートマトンに変換する手法によって ShEx における Property Path を用いた問合せの充足可能性判定を行うアルゴリズムを提案した．具体的には，オートマトンの等価性判定のアルゴリズムをグラフスキーマのオートマトンに応用した．また，評価実験として，グラフデータとそれに妥当な ShEx に対して Ruby 言語を用いて実装した提案アルゴリズムを用いて実行時間を計測し，結果を比較した．その結果，複数用意した充足不能な Property Path のいずれにおいても，グラフデータに対しての探索よりも ShEx に対しての探索の方が時間を短く抑えられることを確認した．

今後の課題としては，本稿では single type semantics のみを対象としたため，multi type semantics にも対応させること，使用できる Property Path のラベルに条件をつけず定義内のあらゆるクエリに対応させること，アルゴリズムの改良などが挙げられる．アルゴリズムの改良案としては，高速化や効率化がある．また，ShEx の仕様には否定などの本稿で扱っていない機能が含まれており，このような機能についても考慮していく必要がある．今後は，これらの課題解決を行なったアルゴリズムを考案予定である．

## 文　　献


[1] World Wide Web Consortium(W3C).“RDF Schema 1.1”. https://www.w3.org/TR/rdf-schema/ , (accessed 2019-10-01).

[2] World Wide Web Consortium (W3C).“Shape Expressions Language 2.next”. https://shexspec.github.io/spec/, (accessed 2019-10-01).

[3] World Wide Web Consortium(W3C).“SPARQL 1.1 Property Paths”. https://www.w3.org/TR/sparql11-property-paths/ , (accessed 2019-10-01).

[4] 松岡茉, 鈴木伸崇.“Shape Expression Schema の下でのパターン問合せ充足可能性判定アルゴリズム”. WebDB Forum 2019 論文集. 2019, pp. 49-52.

[5] Slawek Staworko, Iovka Boneva, José Emilio Labra Gayo, Samuel Hym, Eric G. Prud’hommeaux, Harold R. Solbrig.“Complexity and Expressiveness of ShEx for RDF,” Proc. ICDT, 2015, pp.195-211.

[6] Ken Thompson.“Programming Techniques: Regular expression search algorithm”. Communications of the ACM. 1968, 11(6), pp. 419-422.

[7] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. ”SP2Bench: a SPARQL Performance Benchmark,” Proc. ICDE, 2009, pp. 371-393.

[8] Christian Bizer, Andreas Schultz. ”The Berlin SPARQL Benchmark”. International Journal on Semantic Web & Information Systems, 2009, 5(2), pp. 1-24.

[9] Alex Stolz.“RDF Translator”. http://rdf-translator.appspot.com/ , (accessed 2019-11-15).