Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.12785/amis/080111

A Novel Algorithm for the Conversion of Parallel Regular Expressions to Non-deterministic Finite Automata

Ajay Kumar and Anil Kumar Verma*

Computer Science and Engineering Department, Thapar University, Patiala, Punjab, 147001, India

Received: 11 Mar. 2013, Revised: 16 Jun. 2013, Accepted: 16 Jun. 2013
Published online: 1 Jan. 2014

Abstract: The aim of the paper is to concoct a novel algorithm for the metamorphosis of parallel regular expressions to ε-free non-deterministic finite automata. For a given parallel regular expression r, let m be the number of symbols that occur in r and let C denote the number of concatenation operators in r. In the worst case, 2^{m+1} states are required for the construction of the non-deterministic finite automaton using the novel algorithm. In the earlier existing approaches, the number of states of the non-deterministic finite automaton in the worst case is equal to $2^{2|r|-3C}$.

Keywords: Non-deterministic finite automaton, Parallel finite automaton, Regular expression, Shuffle operator

1 Introduction

Regular expressions(*REs*) play a prominent role in numerous applications, such as finite state based testing, *XML* schema languages, compiler designs, pattern matching and as a programming tool for various scripting languages, such as *Perl* and *PHP* [2,4,10,15]. *REs* used in these applications include union, Kleene closure and concatenation operators. Parallel regular expression(*PRE*) consists of Kleene closure(*), union(+), concatenation(.) and shuffle(&) operators [3].

In one form or another, shuffle operators emerge in multifarious forms of Computer Science, namely multipoint communications [11], *XML* schema language *Relax NG* [8], process algebra [9], and the concurrency of processes [3,4,14]. The metamorphosis of *PREs* to *REs* or deterministic finite automata (*DFAs*) render a way of studying the serialization of concurrency [4], as well as the metamorphosis of *Relax NG* to the *XML* schema definition [8]. *PREs* bestow a concise way to depict concurrent and parallel processes.

1.1 Related Work

Gelade [18] reasserted that the double exponential size is required for the metamorphosis of *PREs* to *REs*. Biegler *et al.* [6] worked on the shuffle decomposition and reaffirmed that shuffle decomposition is not unique over the finite sets. Gelade *et al.* [19] calibrate the *REs* with shuffle operator and numerical constraints. They depict the overview of complexity for the decision problems of *XML* schema languages namely *Relax NG*, *DTDs* and *XSDs*. Hovland [5] worked on the unordered concatenation and proposed a membership algorithm for the *REs* with unordered concatenation and numerical constraints. In unordered concatenation, languages can be concatenated in any order. For example $\&(abc,de) = \{abcde, deabc\}$ where & denotes the unordered concatenation. Standard Generalized Markup Language (*SGML*) [5] permits unordered concatenation. Unordered concatenation is a confined form of shuffle.

Estrade *et al.* [4] calibrate the metamorphosis of *PREs* to *DFAs* using parallel finite automata (*PFAs*) and non-deterministic finite automata (*NFAs*) as intermediate steps. *PREs* to *PFAs* metamorphosis can be accomplished by using the modified Thompson′s construction. *PFAs* to *NFAs* metamorphosis require the effacement of $\lambda-$transitions and enumeration of all possible states explicitly [3]. $\lambda-$transitions represent the joining of two languages using a shuffle operator. The *FLAT* tool kit [7] is designed for the metamorphosis of *PREs* to *DFAs* using the intermediate steps of *PFA* and *NFA*.

In the direct conversion method [1,2], *REs* can be converted into *DFAs* . Given *RE* r over an alphabet Σ with length n. Let m depicts the number of instances of

* Corresponding author e-mail: ajayloura@gmail.com

symbols from Σ in r. Table 1 narrates the output and number of states generated by Thompson′s construction, Glushkov automata [17] and the direct conversion method. Clearly the direct conversion method is preferential in terms of the number of states and outputs generated to the Thompson′s construction and Glushkov automata. There is scope for producing svelte finite automata using the direct conversion method [2] for the metamorphosis of *PREs* to *NFAs*.

Table 1: Comparison between direct conversion, Thompson′s construction and Glushkov automata

Direct Conversion [2]	**Thompson′s Construction[12]**	**Glushkov Automata[17]**
DFA	ε-*NFA*	*NFA*
$\|Q\| \leq m+1$	$\|Q\| \leq 2n$	$\|Q\| = m+1$

Impendence of shuffle operators in *REs* offers umpteenth benefits in the area of process algebra, the concurrency of processes, multipoint communications and *XML* schema language. Impetus for this research stems from the applications of *REs* with shuffle operators. In this paper, the authors propose a novel algorithm for the metamorphosis of *PREs* to ε-free *NFAs* without using any intermediate steps. This algorithm is an extension of the direct conversion of *REs* to *DFAs*.

The contents of the paper are organized as follows. In Section 2, we render some preliminary concepts. In Section 3, we present an algorithm for metamorphose of *PREs* to *DFAs*. Section 4 is devoted to the numerical example followed by section 5 consists of results and discussion. In the last section, concluding remarks are given.

2 Preliminaries

Let Σ be a finite non-empty set of symbols called an alphabet. Σ^* [10] is the free monoid generated by concatenation on Σ, including the empty string. A language is a subset of Σ^*. A string w=$a_1a_2...a_n$ is a finite sequence of symbols taken from Σ, having length n. The length of a string w is denoted by $|w|$. Empty string (denoted by ε) does not contain any symbol.The null language (denoted by ϕ) does not contain any string. A natural number *position* is assigned to each occurrence of symbols from an alphabet and # symbols in r. Position number increases from the left side to the right side of the *RE*. Position of symbol $a \in \Sigma$ of r can be enlightened by $position_a(r)$.

Definition 2.1: Language L elucidated by *RE* r over an alphabet Σ can be defined using the following rules:

1. $L(\phi) \leftarrow \{\phi\}$
2. $L(\varepsilon) \leftarrow \{\varepsilon\}$
3. For $a \in \Sigma, L(a) \leftarrow \{a\}$
4. $L(r_1r_2) \leftarrow L(r_1)L(r_2)$
5. $L(r^*) \leftarrow \cup_{k=0}^{k=\infty}\{(L^k)\}$
6. $L(r_1 \cup r_2) \leftarrow L(r_1) \cup L(r_2)$

Definition 2.2: *DFA* [10] is a quintuple $M = (Q, \Sigma, q_0, \delta, F)$, where

1. Q is the set of states.
2. Σ is an alphabet.
3. $q_0 \in Q$ is the starting state.
4. Transition function(δ) is a partial function mapping $Q \times \Sigma \to Q$.
5. Set $F \subseteq Q$ is the set of final states.

Definition 2.3: *NFA* [10] is a quintuple $M = (Q, \Sigma, q_0, \delta, F)$, where

1. Q is the set of states.
2. Σ is an alphabet.
3. $q_0 \in Q$ is the starting state.
4. Transition function(δ) is a partial function mapping $Q \times \Sigma \to 2^Q$.
5. Set $F \subseteq Q$ is the set of final states.

Definition 2.4: Shuffle operator is denoted by &. Shuffling [13] can be formally defined as follows:

1. For $a \in \Sigma, a\&\varepsilon = \varepsilon\&a = a$
2. For $a, b \in \Sigma,\ x, y \in \Sigma^*$
$$ax\&by = a(x\&by) \cup b(ax\&y)$$

Example 2.1: Consider $w_1 = abc$ and $w_2 = xy$, then
$$w_1\&w_2 = \{abcxy, xyabc, axbcy, axybc, axbyc, abxyc, xabcy, xaybc, xabyc, abxcy\}$$

Definition 2.5: If L_1 and L_2 are two languages, then $L_1\&L_2$ is a set consisting of the strings w such that
$$\{w \mid w = x\&y, \exists x \in L_1 \wedge \exists y \in L_2\}$$

Definition 2.6: Language L elucidated by *PRE* r over an alphabet Σ can be defined using the following rules:

1. $L(\phi) \leftarrow \{\phi\}$
2. $L(\varepsilon) \leftarrow \{\varepsilon\}$
3. For $a \in \Sigma, L(a) \leftarrow \{a\}$
4. $L(r_1r_2) \leftarrow L(r_1)L(r_2)$
5. $L(r^*) \leftarrow \cup_{k=0}^{k=\infty}\{(L^k)\}$
6. $L(r_1 \cup r_2) \leftarrow L(r_1) \cup L(r_2)$
7. $L(r_1\&r_2) \leftarrow L(r_1)\&L(r_2)$

Definition 2.7: *PFA* [16] consists of 7-tuple $(\Sigma, Q, q_0, F, N, \delta, \gamma)$, where

1. Σ is an alphabet.
2. $Q \subseteq 2^N$ is a finite set of states.
3. $q_0 \in Q$ is the starting state.
4. Set $F \subseteq N$ is the set of final states.
5. N is a finite non-empty set of nodes.
6. δ is a state transition function defined by
$$Q \times (\Sigma \cup \lambda) \to 2^Q.$$

7. γ is a node transition function defined by
$2^Q \times (\Sigma \cup \lambda) \rightarrow 2^{2^N}$.

Definition 2.8: Syntax tree of *PRE* r is a binary tree in which every node has at most two children and the following conditions holds:

1. $a \in \Sigma, \#$ and $\#_s$ appear as a leaf node.
2. Operators act as an internal node of the syntax tree.
3. In-order traversal of the syntax tree is equivalent to the *PRE*.

Definition 2.9:*Firstpos(r)*[2] is a function that renders a set of positions of the first symbol occurs in the strings defined by *PRE* r. *Firstpos(r)* is inductively calculated as follows:

1. $firstpos(\phi) \leftarrow firstpos(\varepsilon) \leftarrow \{\emptyset\}$
2. If$((a \in \Sigma) \vee (a = \#))$
   $firstpos(a) \leftarrow position(a)$
3. Let $r = (r_i r_j)$
   If $\varepsilon \notin L(r_i)$ then
   $firstpos(r) \leftarrow firstpos(r_i)$
   Else
   $firstpos(r) \leftarrow firstpos(r_i) \cup firstpos(r_j)$
4. Let $r = (r_i + r_j)$ then
   $firstpos(r) \leftarrow firstpos(r_i) \cup firstpos(r_j)$
5. Let $r = r_i^*$ then
   $firstpos(r) \leftarrow firstpos(r_i)$
6. Let $r = (r_i \& r_j)$ then
   $firstpos(r) \leftarrow firstpos(r_i) \cup firstpos(r_j)$

Definition 2.10:*Lastpos(r)*[2] is a function that renders a set of positions of the last symbol occurs in the strings defined by *PRE* r. *Lastpos(r)* is inductively calculated as follows:

1. $lastpos(\phi) \leftarrow lastpos(\varepsilon) \leftarrow \{\emptyset\}$
2. If$((a \in \Sigma) \vee (a = \#))$
   $lastpos(a) \leftarrow position(a)$
3. Let $r = (r_i r_j)$
   If $(\varepsilon \notin L(r_j))$ then
   $lastpos(r) \leftarrow lastpos(r_j)$
   Else
   $lastpos(r) \leftarrow lastpos(r_i) \cup lastpos(r_j)$
4. Let $r = (r_i + r_j)$ then
   $lastpos(r) \leftarrow lastpos(r_i) \cup lastpos(r_j)$
5. Let $r = r_i^*$ then
   $lastpos(r) \leftarrow lastpos(r_i)$
6. Let $r = (r_i \& r_j)$ then
   $lastpos(r) \leftarrow lastpos(r_i) \cup lastpos(r_j)$

Definition 2.11:*Followpos(i)*[2] of a position i consists of a set of positions which can follow the position i in the *RE* r. *Followpos* is inductively calculated as follows:

1. If $((\varepsilon \in L(r_i)) \wedge (j \in lastpos(r_i)))$
   $followpos(j) \leftarrow followpos(j) \cup_{r_i \in r} firstpos(r_i)$
2. If $(i \in lastpos(r_j))$
   $followpos(i) \leftarrow \cup_{r_j r_k} firstpos(r_k)$

Definition 2.12:*Nullable*[2] is a function that returns either true or false value. *Nullable(r)* is inductively calculated as follows:

1. $nullable(\phi) \leftarrow nullable(a) \leftarrow false$
   $nullable(\varepsilon) \leftarrow true$
2. Let $r = r_i^*$ then
   $nullable(r) \leftarrow true$
3. Let $r = (r_i r_j)$
   If $((nullable(r_i) = true) \wedge (nullable(r_j = true)))$
   $nullable(r) \leftarrow true$
   Else
   $nullable(r) \leftarrow false$
4. Let $r = (r_i \& r_j)$
   If $((nullable(r_i) = true) \wedge (nullable(r_j) = true))$
   $nullable(r) \leftarrow true$
   Else
   $nullable(r) \leftarrow false$
5. Let $r = (r_i + r_j)$
   If $((nullable(r_i) = true) \vee (nullable(r_j) = true))$
   $nullable(r) \leftarrow true$
   Else
   $nullable(r) \leftarrow false$

Definition 2.13:*Shuffle* flag at a node of the syntax tree is calculated as follows:

1. $shuffle(\phi) \leftarrow shuffle(\varepsilon) \leftarrow false$
2. Let $((a \in \Sigma) \vee (a = \#))$
   $shuffle(a) \leftarrow false$
3. Let $((r_i \& r_j) \in r)$
   $shuffle(r_i \& r_j) \leftarrow true$
4. Let $r_i^* \in r$ then
   $shuffle(r_i^*) \leftarrow shuffle(r_i)$
5. Let $((r_i + r_j) \in r)$
   If $(shuffle(r_i) = true) \vee (shuffle(r_j) = true)$
   $shuffle(r_i + r_j) \leftarrow true$
   Else
   $shuffle(r_i + r_j) \leftarrow false$
6. Let $(r_i r_j \in r)$
   If $(shuffle(r_j) = true)$
   $shuffle(r_i r_j) \leftarrow true$
   Else
   If $((shuffle(r_i) = true) \wedge (nullable(r_j) = true))$
   $shuffle(r_i r_j) \leftarrow true$
   Else
   $shuffle(r_i r_j) \leftarrow false$

Example 2.2: Given a particular node with the *PRE* formed at that node.

1. $shuffle(a) \leftarrow false$
2. $shuffle(a \& b) \leftarrow true$
3. $shuffle(a \& b)^* \leftarrow true$
4. $shuffle(ab)^* \leftarrow false$
5. $shuffle(a \& b)c \leftarrow false$
6. $shuffle(a(b \& c)) \leftarrow true$
7. $shuffle(a \& b)c^* \leftarrow true$

Definition 2.14:*Shuffle-first(r)* of a *PRE* r consists of set of positions calculated as follows:

1. $shuffle-first(r) \leftarrow \{\emptyset\}$
2. If $((r_i\#_s \& r_j\#_s) \in r)$
   $shuffle-first(r) \leftarrow shuffle-first(r) \cup$
   $Min(firstpos(r_i\#_s)) \cup Min(firstpos(r_j\#_s))$

Definition 2.15:*Shuffle-last(r)* of a *PRE* r consists of set of positions calculated as follows:

1. $shuffle-last(r) \leftarrow \{\emptyset\}$
2. If $((r_i\#_s \& r_j\#_s) \in r)$
   $shuffle-last(r) \leftarrow shuffle-last(r) \cup$
   $Max(lastpos(r_i\#_s)) \cup Max(lastpos(r_j\#_s))$

Example 2.3: Given *PRE* $r = ((ab)\#_s \& (cd)\#_s)\#$
Augmented *PRE* with position $r' = ((ab)\#_s \& (cd)\#_s)\#$
$shuffle-first(r) \leftarrow shuffle-first(r) \cup$
$Min(firstpos(ab\#_s)) \cup Min(firstpos(cd\#_s)) \leftarrow \{1,4\}$
$shuffle-last(r) \leftarrow shuffle-last(r) \cup$
$Max(lastpos(ab\#_s)) \cup Max(lastpos(cd\#_s)) \leftarrow \{3,6\}$

Followpos of $\#_s$ are taken as $\emptyset$. Now the question arises: which one is the next symbol to be processed, after reading each symbol of r_1 and r_2 involved in shuffling? Immediate-follow will give the positions to be followed after a pair of $\#_s$ involved in the shuffling.

Definition 2.16:*Immediate-follow(p)* of a set of positions p labeled by $\#_s$ is calculated as follows:

1. Let $((r_i\#_s \& r_j\#_s) r_k \in r)$
   If $((\varepsilon \in r_i) \wedge (\varepsilon \in r_j))$
   $IF \leftarrow position_{\#_s}(r_i\#_s) \cup position_{\#_s}(r_j\#_s)$
   $immediate-follow(IF) \leftarrow immediate-follow(IF)$
   $\cup firstpos(r_i\#_s) \cup position_{\#_s}(r_j\#_s) \cup firstpos(r_k)$
2. Let $((r_i\#_s \& r_j\#_s) r_k \in r)$
   If $((\varepsilon \in r_j\#_s) \wedge (\varepsilon \notin r_i))$
   $IF \leftarrow position_{\#_s}(r_i\#_s) \cup position_{\#_s}(r_j\#_s)$
   $immediate-follow(IF) \leftarrow immediate-follow(IF)$
   $\cup firstpos(r_j\#_s) \cup position_{\#_s}(r_i\#_s) \cup firstpos(r_k)$
3. Let $((r_i\#_s \& r_j\#_s) r_k \in r)$
   $IF \leftarrow position_{\#_s}(r_i\#_s) \cup position_{\#_s}(r_j\#_s)$
   $immediate-follow(IF) \leftarrow immediate-follow(IF)$
   $\cup firstpos(r_k)$
4. Let $((r_i\#_s \& r_j\#_s)^* r_k \in r)$
   $IF \leftarrow position_{\#_s}(r_i\#_s) \cup position_{\#_s}(r_j\#_s)$
   $immediate-follow(IF) \leftarrow immediate-follow(IF)$
   $\cup firstpos(r_i \& r_j)$
   $immediate-follow(IF) \leftarrow immediate-follow(IF)$
   $\cup firstpos(r_k)$

Example 2.4: Given the *PRE* $r \leftarrow ((ab)\#_s \& (cd)\#_s)e\#$
Augmented *PRE* with their position is $((a_1b_2)\#_{s3} \& (c_4d_5)\#_{s6})e_7\#_8$
$immediate-follow(3,6) \leftarrow \{7\}$

Example 2.5: Given the *PRE* $r \leftarrow ((ab)^*\#_s \& (cd)\#_s)e\#$
Augmented *PRE* with their position is $((a_1b_2)^*\#_{s3} \& (c_4d_5)\#_{s6})e_7\#_8$
$immediate-follow(3,6) \leftarrow \{1,6,7\}$

Example 2.6: Given the *PRE* $r \leftarrow ((ab)\#_s \& (cd)\#_s)^* e\#$
Augmented *PRE* with their position is $((a_1b_2)\#_{s3} \& (c_4d_5)\#_{s6})^* e_7\#_8$
$immediate-follow(3,6) \leftarrow \{1,4,7\}$

Example 2.7: Given the *PRE* $r \leftarrow ((ab)^*\#_s \& (cd)^*\#_s)e\#$
Augmented *PRE* with their position is $((a_1b_2)^*\#_{s3} \& (c_4d_5)^*\#_{s6})e_7\#_8$
$immediate-follow(3,6) \leftarrow \{1,4,7\}$

Definition 2.17:*Separator* is a functions that returns either true or false value. Separator at a node n is inductively calculated as follows:

1. $separator(\phi) \leftarrow separator(\varepsilon) \leftarrow separator(a) \leftarrow separator(\#) \leftarrow false$
2. Let $(r_i + r_j) \in r$
   If$((shuffle(r_i) = true) \vee (shuffle(r_j) = true))$
   $separator(r_i + r_j) \leftarrow true$
   Else
   $separator(r_i + r_j) \leftarrow false$
3. Let $(r_i^*) \in r$
   $separator(r_i^*) \leftarrow separator(r_i)$
4. Let $(r_i \& r_j) \in r$
   If$((shuffle(r_i) = true) \vee (shuffle(r_j) = true))$
   $separator(r_i \& r_j) \leftarrow true$
   Else
   $separator(r_i \& r_j) \leftarrow false$
5. Let $(r_i r_j) \in r$
   If$((shuffle(r_i) = true) \wedge (shuffle(r_j) = true))$
   If $(nullable(r_j) = true)$
   $separator(r_i r_j) \leftarrow true$
   Else
   $separator(r_i r_j) \leftarrow false$

Definition 2.18:*Separator-first(r)* of a *PRE* r is calculated as follows:

1. $Separator-first(r) \leftarrow \emptyset$
2. Let $(r_i | r_j) \in r$
   If$((shuffle(r_i) = true) \vee (shuffle(r_j) = true))$
   $Separator-first(r) \leftarrow Separator-first(r) \cup$
   $Min(firstpos(r_i)) \cup Min(firstpos(r_j))$
3. Let $(r_i r_j) \in r$
   If$((shuffle(r_i) = true) \wedge (shuffle(r_j) = true) \wedge$
   $(nullable(r_j) = true))$
   $Separator-first(r) \leftarrow Separator-first(r) \cup$
   $Min(firstpos(r_i)) \cup Min(firstpos(r_j))$

Definition 2.19:*Separator-last(r)* of a *PRE* r is calculated as follows:

1. $Separator-last(r) \leftarrow \emptyset$
2. Let $(r_i | r_j) \in r$
   If$((shuffle(r_i) = true) \vee (shuffle(r_j) = true))$
   $Separator-last(r) \leftarrow Separator-last(r) \cup$
   $Max(lastpos(r_i)) \cup Max(lastpos(r_j))$
3. Let $(r_i r_j) \in r$
   If$((shuffle(r_i) = true) \wedge (shuffle(r_j) = true) \wedge$

$$(nullable(r_j) = true))$$
$$Separator - last(r) \leftarrow Separator - last(r) \cup$$
$$Max(lastpos(r_i)) \cup Max(lastpos(r_j))$$

Example 2.8: Given the augmented *PRE* $r = ((a_1\#_{s2})\&(b_3\#_{s4}))|((c_5\#_{s6})\&(d_7\#_{s8}))$ at a node n.
$separator(n) \leftarrow true$
$separator - first(r) \leftarrow \{1,5\}$
$separator - last(r) \leftarrow \{4,8\}$

Example 2.9: Given the augmented *PRE* $r = ((a_1\#_{s2})\&(b_3\#_{s4})).((c_5\#_{s6})\&(d_7\#_{s8}))$ at a node n.
$separator(n) \leftarrow true$
$separator - first(r) \leftarrow \{1,5\}$
$separator - last(r) \leftarrow \{4,8\}$

3 Proposed Algorithm

In the direct conversion [1,2] of *RE* to *DFA*, a syntax tree is constructed conforming to the augmented *RE r#* such that all operators act as the internal nodes and symbols act as the leaf node of the syntax tree. Positions are assigned to the symbols of the *RE* increasing from left to right. The leftmost symbol of r is assigned the position one. *Firstpos*, *lastpos* and *nullable* are defined for all the nodes of the syntax tree. *Followpos* is computed for the leaf node of the syntax tree. Using *firstpos*, *lastpos*, *nullable* and *followpos*, an equivalent *DFA* is generated.

In this section, the authors propose the algorithm (*MPRENFA*) for the metamorphosis of *PRE* to *NFA*. This conversion is evolved on the followpos of symbols of the *PRE*. The detail of the *MPRENFA* algorithm is as follows:

1. **Augmented PRE:** Add # at the end of PRE. If $(r_i\&r_j) \in r$ enclose $\#_s$ after r_i and r_j. Assign position to the symbols of the augmented *PRE* from left to right.

2. **Calculation of Followpos, Shuffle-first, Shuffle-last and Immediate-follow:** Syntax tree is constructed for the augmented *PRE*. *L* and *R* enlighten the left and the right child of a node during traversal of the syntax tree. Firstpos, lastpos, nullable and followpos of the nodes are determined using definitions 2.9 to 2.12 with meagre modification. These variations occur due to the shuffle operators. Shuffle, shuffle-first, shuffle-last, separator-first, separator-last, separator and immediate-follow are determined using the definitions 2.13 to 2.19.

3. ***NFA* Creation:** Using different values determined in step 2, ε-*NFA* is generated.

Shuffle operator is associative. Given *PRE* $r = r_1\&r_2\&r_3$ is converted into $(r_1\&r_2)\&r_3$ or $r_1\&(r_2\&r_3)$ before applying the algorithm.

Example 3.1: Given *PRE* $r = ((a\&b)^*c)\&d$

Using step 1 and step 2 of the algorithm *MPRENFA*,

Augmented PRE $r' = ((a_1\#_{s2}\&b_3\#_{s4})^*c_5\#_{s6})\&d_7\#_{s8}\#_9$

A syntax tree as shown in Fig. 1 is constructed for the augmented *PRE* r'.During traversal of the syntax tree in post-order, different values are computed using the definitions 2.8 to 2.18.

Notation used in the algorithm : Current node of the post-order traversal is represented by n. The left and right children are depicted by L and R respectively. At any instant of time, let *sf* indicates the number of elements in the separator-first. Based on the values of separator-first and separator-last, immediately-follow is determined in step 5 and 6 of the procedure Follow_Calc. In step 4 and 5 of the procedure shuffle-traverse, for avoiding duplicate entries in the shuffle-first, care should be taken so that the left and right branch of the syntax tree, which was already traversed, does not contain a shuffle operator. *Min* and *Max* represent the minimum and maximum value from a set. The complete procedure is explained with the help of example 3.2. *symbol(i)* depicts the symbol at i^{th} position. *IF* depicts the positions of a pair of $\#_s$.

Example 3.2: On applying $Follow\_Calc(r')$ to the Example 3.1

Using algorithm 2, following values are determined during traversal of the syntax tree:

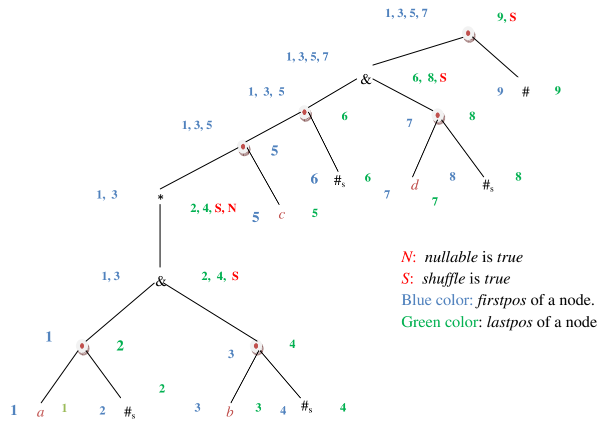


Fig. 1: Syntax tree for the augmented *PRE* $r = ((a\#_s\&b\#_s)^*c\#_s)\&d\#_s\#$

$$immediate - follow(2,4) \leftarrow \{1,3,5\}$$
$$immediate - follow(6,8) \leftarrow \{9\}$$
$$shuffle - first \leftarrow \{1,3\}$$
$$shuffle - last \leftarrow \{2,4\}$$
$$followpos(1) \leftarrow \{2\}\ ,\ followpos(3) \leftarrow \{4\}$$
$$followpos(5) \leftarrow \{6\}\ ,\ followpos(7) \leftarrow \{8\}$$

Using different values determined from algorithm *Follow_Calc*, we can convert *PRE* into $\varepsilon-$free *NFA*. The novel algorithm for the construction of *NFA* is given in algorithm 5. Consider the state $q \in Q$ consists of positions from the sub-expression r_i and r_j such that $(r_i\&r_j) \in r$. Consider the situation, where we have to take the symbol a on q from r_i then *epos* depicts the positions in q pertaining to r_j. Epos will not include the position of $\#_s$.

Lpos depicts the position on which we have recently read the symbol a. $q \in Q$ depicts the current states on which symbol are read. Consider there are n entries in the shuffle-first and shuffle-last. Consider there are m entries in the separator-first and separator-last. The step by step procedure of the algorithm is illustrated in section 4 by a numerical example.

Input: *PRE r*
Output: $\varepsilon-$free *NFA M* such that *L(M)=L(r)*
initialization;

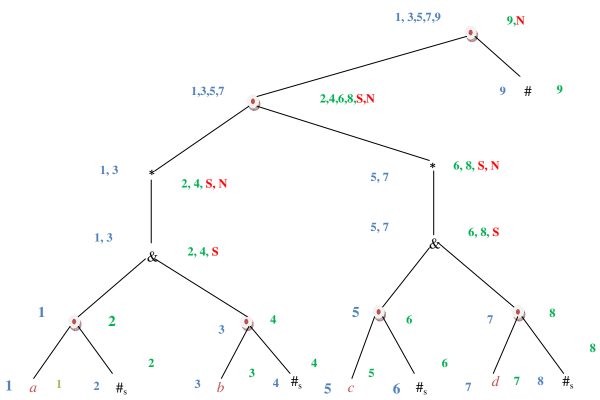
1. **for** $\exists((r_i \& r_j) \in r)$ **do** ▷ Augmented *PRE*
   $r_i \leftarrow r_i\#_s$
   $r_j \leftarrow r_j\#_s$
   end
2. $r' \leftarrow r\#$
3. Assign position to the symbols of the augmented *PRE*
4. Follow_Calc(r') ▷ Call to procedure Follow_Calc
5. Create-NFA() ▷ Call to procedure Create-NFA

Algorithm 1: *MPRENFA(r,NFA)*

4 Numerical Example

In this section, the complete procedure for the conversion of *PRE* to *NFA* is described with the help of an example.

Example 4.1: Given the *PRE* $r = (a\&b)^*(c\&d)^*$



N denotes the *nullable* is *true* and *S* denotes the *shuffle* is *true*.
Blue color values denote the *firstpos* of a node.
Green color values denote the *lastpos* of a node.

Fig. 2: Syntax tree for the augmented *PRE* $r = (a\#_s\&b\#_s)^*(c\#_s\&d\#_s)^*$ using the proposed approach

Fig. 2 delineates the syntax tree corresponding to r. $followpos(1) \leftarrow \{2\}, followpos(3) \leftarrow \{4\}$

Input: *An Augmented PRE* r'
Output: $Followpos, shuffle-first, shuffle-last, immediate-follow, separator-first, separator-last$
initialization;

1. Construct a syntax tree for r'
2. Repeat step 3 to 7 for each node n during the post-order traversal of the syntax tree ▷ Leaf Node
3. **if** $(((n=a) \wedge (a \in \Sigma)) \vee ((n=\#) \vee (n=\#_s)))$ **then**
   $firstpos(n) \leftarrow lastpos(n) \leftarrow position(n)$
   $nullable(n) \leftarrow shuffle(n) \leftarrow false$
   $separator(n) \leftarrow false$
   end
   ▷ Union operator
4. **if** $n = +$ **then**
   $firstpos(n) \leftarrow firstpos(L) \cup firstpos(R)$
   $lastpos(n) \leftarrow lastpos(L) \cup lastpos(R)$
   if $((nullable(L) = false) \wedge (nullable(R) = false))$ **then**
   $nullable(n) \leftarrow false$
   else
   $nullable(n) \leftarrow true$
   end
   if $((shuffle(L) = true) \vee (shuffle(R) = true))$ **then**
   $shuffle(n) \leftarrow separator(n) \leftarrow true$
   $separator-first \leftarrow separator-first \cup Min(firstpos(L)) \cup Min(firstpos(R))$
   $separator-last \leftarrow separator-last \cup Max(firstpos(L)) \cup Max(firstpos(R))$
   else
   $shuffle(n) \leftarrow false$
   $separator(n) \leftarrow false$
   end
   end
   ▷ Concatenation operator, call to Algorithm 3
5. **if** $n = .$ **then**
   Concatenation-traverse();
   end
   ▷ Kleene closure
6. **if** $n = *$ **then**
   $firstpos(n) \leftarrow firstpos(L)$
   $lastpos(n) \leftarrow lastpos(L)$
   $nullable(n) \leftarrow true$
   $separator(n) \leftarrow separator(L)$
   $shuffle(n) \leftarrow shuffle(L)$
   $IF \leftarrow \emptyset$
   for $i \in lastpos(n)$ **do**
   if $(symbol(i) \neq \#_s)$ **then**
   $followpos(i) \leftarrow followpos(i) \cup firstpos(n)$
   else
   $IF \leftarrow IF \cup i$
   end
   end
   if $(IF \neq \emptyset)$ **then**
   if $separator(n) = true$ **then**
   ▷ Assume sf elements in separator-first
   for $(i \leftarrow 1; i \leq sf; i \leftarrow i+1)$ **do**
   for $(z \in IF)$ **do**
   $T \leftarrow \emptyset$
   if $separator-first[i] \leq z \leq separator-last[i]$ **then**
   $T \leftarrow T \cup z$
   end
   $immediate-follow(IF) \leftarrow firstpos(L) \cup immediate-follow(IF)$
   end
   end
   else
   $immediate-follow(IF) \leftarrow immediate-follow(IF) \cup firstpos(L)$
   end
   end
   end
   ▷ Shuffle operator, call to Algorithm 4
7. **if** $n = \&$ **then**
   Shuffle-traverse();
   end

Algorithm 2: *Procedure Follow_Calc(r')*

```
Input: An Augmented PRE r′
Output: Followpos, shuffle−first,
        shuffle−last, immediate−follow,
        separator−first, separator−last
        at Concatenation operator.
initialization;
1. firstpos(n) ← firstpos(L)
   lastpos(n) ← lastpos(R)
2. if nullable(L) = true then
     firstpos(n) ← firstpos(n) ∪ firstpos(R)
   end
3. if nullable(R) = true then
     lastpos(n) ← lastpos(n) ∪ lastpos(L)
   end
4. if (nullable(L) = true) ∧ ((nullable(R) = true) ∨ (R = #s))
   then
     nullable(n) ← true
   else
     nullable(n) ← false
   end
5. if (shuffle(R) = true) then
     shuffle(n) ← true
   else
     if ((shuffle(L) = true) ∧ ((R = #s) ∨
         (nullable(R) = true))) then
         shuffle(n) ← true
     else
         shuffle(n) ← false
     end
   end
6. if ((shuffle(L) = true) ∧ (shuffle(R) = true)) then
     if nullable(R) = true then
         separator(n) ← true
         separator−first ← separator−first ∪
               Min(firstpos(L)) ∪ Min(firstpos(R))
         separator−last ← separator−last ∪
               Max(lastpos(L)) ∪ Max(lastpos(R))
     end
   else
     separator(n) ← false
   end
                                  ▷ IF consists of positions of #s
7. IF ← ∅
   for i ∈ lastpos(L) do
     if (symbol(i) ≠ #s) then
         followpos(i) ← followpos(i) ∪ firstpos(R)
     else
         IF ← IF ∪ i
     end
   end
8. if IF ≠ ∅ then
                      ▷ Assume sf elements in separator-first
     if separator(n) = true then
         for (i = 1; i = sf; i = i+1) do
             for z ∈ IF do
                 T ← ∅
                 if separator−first[i] ≤ z ≤
                   separator−last[i] then
                     T ← T ∪ z
                 end
             end
             immediate−follow(T) ← firstpos(R) ∪
             immediate−follow(T)
         end
     else
         immediate−follow(IF) ← ∪ firstpos(R)
         immediate−follow(IF)
     end
   end
```

Algorithm 3: *Procedure Concatenation-traverse*

```
Input: An Augmented PRE r′
Output: Followpos, shuffle−first,
        shuffle−last, immediate−follow,
        separator−first, separator−last
        at Shuffle operator.
initialization;
1. firstpos(n) ← firstpos(L) ∪ firstpos(R)
   lastpos(n) ← lastpos(L) ∪ lastpos(R)
   shuffle(n) ← true
2. if (nullable(L) = true) ∧ (nullable(R) = true) then
     nullable(n) ← true
   else
     nullable(n) ← false
   end
3. flag ← false
                    ▷ For avoiding duplicate shuffle-first entry
   if (shuffle(L) = false) then
                          ▷ Assume m shuffle-first entry
     for (z = 1; z < m; z = z+1) do
         if (Min(firstpos(L)) ≤ shuffle−first[z] ≤
            Max(lastpos(L))) then
             flag ← true
             break
         end
     end
     if (flag ≠ true) then
         shuffle−first ← shuffle−first ∪ Min(firstpos(L))
         shuffle−last ← shuffle−last ∪ Max(lastpos(L))
     end
   end
4. flag ← false
                    ▷ For avoiding duplicate shuffle-first entry
   if (shuffle(R) = false) then
                          ▷ Assume m shuffle-first entry
     for (z = 1; z < m; z = z+1) do
         if (Min(firstpos(R)) ≤ shuffle−first[z] ≤
            Max(lastpos(R))) then
             flag ← true
             break
         end
     end
     if (flag ≠ true) then
         shuffle−first ← shuffle−first ∪
             Min(firstpos(R))
         shuffle−last ← shuffle−last ∪ Max(lastpos(R))
     end
   end
5. if (separator(L) = true) ∨ (separator(R) = true) then
     separator(n) ← true
   end
                               ▷ determine immediate-follow
6. IF ← ∅
   if (i ∈ lastpos(n)) then
     if (symbol(i) = #s) then
         IF ← IF ∪ i
     end
   end
   if (IF ≠ ∅) then
     immediate−follow(IF) ← ∅
   if (nullable(L) = true) then
     immediate−follow(IF) ← firstpos(L) ∪
         immediate−follow(IF)
     if (nullable(R) = false) then
         immediate−follow(IF) ← firstpos(R) ∪
         immediate−follow(IF)
     end
   end
   if (nullable(R) = true) then
     immediate−follow(IF) ← firstpos(R) ∪
         immediate−follow(IF)
     if (nullable(L) = false) then
         immediate−follow(IF) ← firstpos(L) ∪
             immediate−follow(IF)
     end
   end
   end
```

Algorithm 4: *Procedure Shuffle-traverse*

Input: *AugmentedPRE* and different values calculated using $Follow\_Calc(r)$

Output: An equivalent *NFA* corresponding to *PRE*

1. Starting state $q_0 \leftarrow firstpos(root)$
   $Q \leftarrow \{q_0\}$ and make q_0 as unmarked state.
2. **for** $\exists(unmarked\ q \in Q)$ **do**
   Choose an unmarked state q and mark it.
   3. **for** $\exists a \in \Sigma$ **do**
      ▷ for each value of shuffle-first and shuffle-last
      4. **for** $i = 1\ to\ i = n$ **do**
         $epos \leftarrow lpos \leftarrow qtrans \leftarrow \emptyset$
         for $\exists j \in q$ **do**
            if $shuffle - first[i] \leq j \leq shuffle - last[i]$ **then**
               if $symbol(j) = a$ **then**
                  $qtrans \leftarrow qtrans \cup followpos(j)$
                  $lpos \leftarrow j$
               else
                  if $symbol(j) \neq \#$ **then**
                     $epos \leftarrow epos \cup j$
                  end
               end
            else
            end
         end
         if $qtrans = \emptyset$ **then**
            go to 4
         end
         for $j = 1\ to\ j = m$ **do**
            if $(separator - first[j] \leq lpos \leq separator - last[j]$ **then**
               break
            end
         end
         for $\exists pos \in epos$ **do**
            if $(separator - first[j] > pos) \vee (separator - last[j] < pos)$ **then**
               $epos = epos - pos$
            end
         end
         $qtrans \leftarrow qtrans \cup epos$
         $IF \leftarrow \emptyset$
         for $\exists pos \in qtrans$ **do**
            if $(symbol(pos) = \#_s)$ **then**
               $IF \leftarrow IF \cup pos$
            end
         end
         if $(immediate - follow(IF) \neq \emptyset)$ **then**
            $qtrans \leftarrow qtrans - IF \cup immediate - follow(IF)$
            $trans[q,a] = qtrans$
            $Q \leftarrow Q \cup qtrans$
         end
      end
   end
end
5. Final state:
   $F = \{\forall q_f | lastpos(root) \subseteq q_f \wedge q_f \in Nstates\}$

Algorithm 5: *Procedure Create-NFA*

$followpos(5) \leftarrow \{6\}, followpos(7) \leftarrow \{8\}$
$shuffle - first(r) \leftarrow \{1,3,5,7\}$
$shuffle - last(r) \leftarrow \{2,4,6,8\}$
$immediate - follow(2,4) \leftarrow \{1,3,5,7,9\}$
$immediate - follow(6,8) \leftarrow \{5,7,9\}$
$separator - first \leftarrow \{1,5\}$
$separator - last \leftarrow \{4,8\}$

NFA creation:

$q_0 \leftarrow firstpos(root) \leftarrow \{1,3,5,7,9\}$

Reading of symbols on q_0

Symbol a
$i \leftarrow 1$
$qtrans \leftarrow followpos(1) \leftarrow \{2\}$
$epos \leftarrow \{3,5,7\}$
Using $separator - first \leftarrow \{1,5\}$
$separator - last \leftarrow \{4,8\}, epos \leftarrow \{3\}$
$qtrans \leftarrow followpos(1) \cup epos \leftarrow \{2,3\}$
$trans(q_0,a) \leftarrow \{2,3\} \leftarrow q_1$
$i \leftarrow 2, i \leftarrow 3, i \leftarrow 4$
$qtrans \leftarrow \emptyset$

Symbol b
$i \leftarrow 1$
$qtrans \leftarrow \emptyset$
$i \leftarrow 2$
$qtrans \leftarrow followpos(3) \leftarrow \{4\}, epos \leftarrow \{1,5,7\}$
Using $separator - first \leftarrow \{1,5\}$,
$separator - last \leftarrow \{4,8\}, epos \leftarrow \{1\}$
$qtrans \leftarrow followpos(3) \cup epos \leftarrow \{1,4\}$
$trans(q_0,b) \leftarrow \{1,4\} \leftarrow q_2$
$i \leftarrow 3, i \leftarrow 4$
$qtrans \leftarrow \emptyset$

Symbol c
$i \leftarrow 1, i \leftarrow 2$
$qtrans \leftarrow \emptyset$
$i \leftarrow 3$
$qtrans \leftarrow followpos(5) \leftarrow \{6\}, epos \leftarrow \{1,3,7\}$
Using $separator - first \leftarrow \{1,5\}$,
$separator - last \leftarrow \{4,8\}, epos \leftarrow \{7\}$
$qtrans \leftarrow followpos(5) \cup epos \leftarrow \{6,7\}$
$trans(q_0,c) \leftarrow \{6,7\} \leftarrow q_3$
$i \leftarrow 4$
$qtrans \leftarrow \emptyset$

Symbol d
$i \leftarrow 1, i \leftarrow 2, i \leftarrow 3$
$qtrans \leftarrow \emptyset$
$i \leftarrow 4$
$qtrans \leftarrow followpos(7) \leftarrow \{8\}, epos \leftarrow \{1,3,5\}$
Using $separator - first \leftarrow \{1,5\}$,
$separator - last \leftarrow \{4,8\}, epos \leftarrow \{5\}$
$qtrans \leftarrow followpos(7) \cup epos \leftarrow \{5,8\}$
$trans(q_0,d) \leftarrow \{5,8\} \leftarrow q_4$

Reading of symbols on q_1

Symbol a
$i \leftarrow 1$
$qtrans \leftarrow followpos(1) \leftarrow \{2\}, epos \leftarrow \{4\}$
Using $separator - first \leftarrow \{1,5\}$,
$separator - last \leftarrow \{4,8\}, epos \leftarrow \{4\}$
$qtrans \leftarrow followpos(1) \cup epos \leftarrow \{2,4\}$

$trans(q_2,a) \leftarrow immediate - follow\{2,4\} \leftarrow \{1,3,5,7,9\} \leftarrow q_0$
$i \leftarrow 2, i \leftarrow 3, i \leftarrow 4$
$qtrans \leftarrow \emptyset$
Symbol b
$i \leftarrow 1$
$qtrans \leftarrow \emptyset$
$i \leftarrow 2$
$qtrans \leftarrow followpos(3) \leftarrow \{4\}, epos \leftarrow \{2\}$
Using $separator - first \leftarrow \{1,5\}$,
$separator - last \leftarrow \{4,8\}, epos \leftarrow \{2\}$
$qtrans \leftarrow followpos(3) \cup epos \leftarrow \{2,4\}$
$trans(q_1,b) \leftarrow immediate - follow\{2,4\} \leftarrow \{1,3,5,7,9\} \leftarrow q_0$
Symbol c
$i \leftarrow 1, i \leftarrow 2, i \leftarrow 3, i \leftarrow 4$
$qtrans \leftarrow \emptyset$
Symbol d
$i \leftarrow 1, i \leftarrow 2, i \leftarrow 3, i \leftarrow 4$
$qtrans \leftarrow \emptyset$

On repeating the procedure, we obtain the *NFA* $M(\{q_0,q_1,q_2,q_3,q_4,q_5\},\{a,b\},q_0,\delta,\{q_0,q_5\})$shown in Fig. 3.

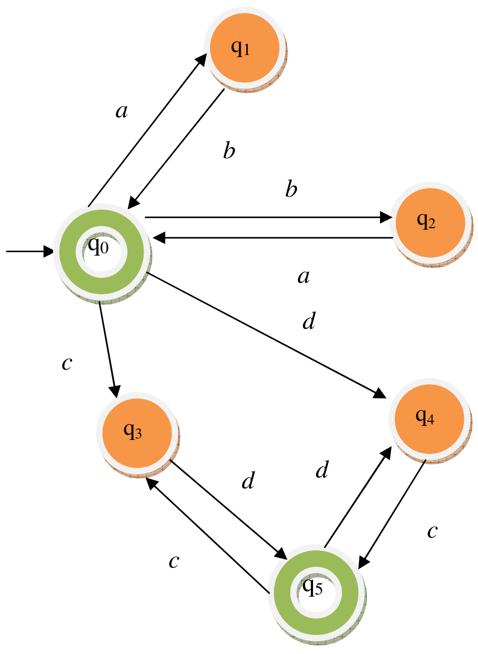


Fig. 3: *NFA* for $PREr = (a\&b)^*(c\&d)^*$

5 Results and Discussion

Using Estrade *et al.*$'$s methodology [4], *PREs* r can be converted into *PFAs* using the modified Thompson$'$s construction requiring $2|r| - 3C$ where C is the number of times a concatenation operator appears in r. A *PFA* (having $2|r| - 3C$ states) can be converted into *NFA* using subset construction requiring $2^{2|r|-3C}$ states in the worst case. Fig. 4 delineates the *PFA* corresponding to *PRE* $a\&b^*$ using Estrade *et al.*$'$*s* methodology. Fig. 5 delineates the *DFA* corresponding to the *PRE* $a\&b^*$ using the proposed algorithm. Table 2 depicts the differences between the proposed algorithm and Estrade *et al.*$'$*s* methodology.

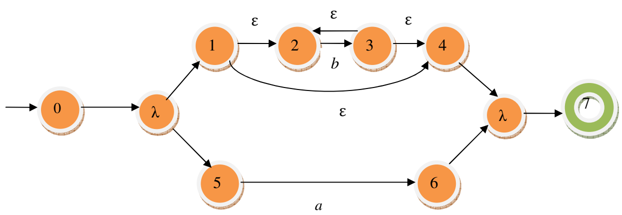


Fig. 4: PFA for $r = a\&b^*$ using Estrade *et al.*$'$s methodology

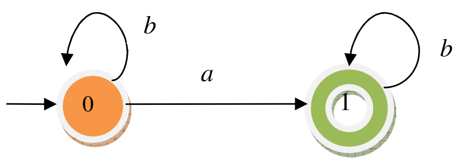


Fig. 5: DFA for $r = a\&b^*$ using the proposed approach

Table 2: Comparison between Estrade *et al.*'s methodology and proposed algorithm

Estrade et al.[4]	Proposed Algorithm
ε-*NFA*	ε-*free NFA*
$PRE \rightarrow PFA \rightarrow \varepsilon - NFA$	$PRE \rightarrow NFA$
$2^{2\|r\|-3C}$ states	2^{m+1}

Theorem 5.1: Let m denote the total number of occurrences of symbols in *PRE* r, then the worst case state complexity of the *NFA* generated using the proposed approach is equal to 2^{m+1}.

Proof. A state of the *NFA* can be constructed from a set of positions. Maximum 2^n states can be constructed using the n positions. The number of leaf nodes in the syntax tree is equal to $m+2s+1$. A state except the final state cannot be constructed by contemplating only the positions of $\#_s$. The number of positions is equal to $(m+1)$ by excluding the positions of $\#_s$. Hence the worst case state complexity of the NFA constructed from $(m+1)$ positions is equal to 2^{m+1} states. □

6 Conclusions and Future Scope

An algorithm is proposed for the metamorphosis of *PREs* to ε-*NFAs*. The worst case state complexity of the *NFA* is equal to 2^{m+1} which is a significant improvement over the Estrade *et al.* approach [4]. The major benefit of the novel algorithm is the production of a svelte *NFA*. Another major benefit of the novel algorithm is that the *PREs* can be converted into ε-*free NFAs* without using any intermediate steps. In future, work can be done on reducing the time complexity of the proposed algorithm. A tool can be designed for the conversion of *PRE* to *NFA* using the proposed algorithm.

Acknowledgement

The authors are grateful to the anonymous referee for a careful checking of the details and for helpful comments and suggestion that significantly improve the quality of this paper.

References

[1] A. Brueggemann-Klein, Regular expressions into finite automata, *Theoretical Computer Science*, **120**, 197-213 (1993).
[2] A. V. Aho, R. Sethi and J. D. Ullman, Compilers: principles, techniques and tools, *Pearson Education*, (2005).
[3] B. D. Estrade, An investigation of equivalent serialized forms of parallel finite automata, *Master′s Thesis, The University of Southern Mississippi*, (2005).
[4] B. D. Estrade, A. L. Perkins and J. M. Harris, Explicitly parallel regular expressions, *Proc. of the first International multi-symposiums on computer and computational sciences (IMSCCS06), IEEE*, 402-409 (2006).
[5] D. Hovland, The membership problem for regular expressions with unordered concatenation and numerical constraints, *Language, Automata Theory and Applications, Lecture Notes in Computer Science*, **7183**, 313-324 (2012).
[6] F. Biegler, M. Daley, M. Holzer and I. McQuillan, On the uniqueness of shuffle on words and finite languages, *Theoretical Computer Science*, **410**, 3711-3724 (2009).
[7] http://www.0x743.com/flat.
[8] J. Clark and M. Murata, RELAX NG specifications, Available: http://relaxng.org/spec-20011203.html, (2001).
[9] J. C. M. Baeten and W. P. Weijland, Process algebra, *Cambridge tract in theoretical computer science, Cambridge University press, Cambridge*, **18**, (1990).
[10] J. E. Hopcroft, R. Motwani and J. D. Ullman, Introduction to automata theory, Languages and computation, *Pearson Education*, (2005).
[11] K. Iwama, Unique decomposability of shuffled strings: A formal treatment of asynchronous time-multiplexed communication, *Proc. of the 15th annual ACM symposium on theory of computation*, 374-381 (1983).
[12] K. Thompson, Regular expression search algorithm, *Communications of the ACM*, **11**, 419-422 (1968).
[13] M. H. Ter Beek and J. Kleijn, Infinite unfair shuffles and associativity, *Theoretical Computer Science*, **380**, 401-410 (2007).
[14] Nivat M., Behaviours of synchronized systems of processes, *L.I.T.P. Report no. 81-64 University de Paris 7*, (1981).
[15] P. D. Stotts and W. Pugh, Parallel finite automata for modeling concurrent software systems, *Journals of Systems and Software*, **27**, 27-43 1994.
[16] V. K. Garg, Modeling of distributed systems by concurrent regular expressions, *Proc. of the* 2^{nd} *International conference on formal description techniques for distributed systems and communication protocols*, (1989).
[17] V. M. Glushkov, The abstract theory of automata, *Russian Mathematical Surveys*, **16**, 1-53 (1961).
[18] W. Gelade, Succinctness of regular expressions with interleaving, intersection and counting, *Theoretical Computer Science*, **411**, 2987-2998 (2010).
[19] W. Gelade, W. Martens and F. Neven, Optimizing schema languages for XML: numerical constraints and interleaving, *Database Theory-ICDT 2007, Lecture Notes in Computer Science Springer*, **4353**, 269-283 (2006).

Ajay Kumar is an Assistant Professor at Computer Science and Engineering Department, Thapar University, Patiala, Punjab, India. He obtained his M. Tech. Computer Science and Engineering from Kurukeshtra University, India in 2004. He has nine years of teaching experience in the area of Theory of Computation, Software Testing and Programming Languages. His research interests are Theoretical Computer Science and Software Testing.

Anil Kumar Verma is Associate Professor at Computer Science and Engineering Department, Thapar University, Patiala, India.He obtained his B.S. and M.S. in 1991 and 2001 respectively, majoring in Computer Science and Engineering. He obtained his PhD in Mobile Adhoc Network from Thapar University, India in 2008. His research interests are wireless networks, routing algorithms, Theoretical Computer Science and cloud computing. He has published more than 100 papers in referred Journals. He has chaired various sessions in the International and National Conferences. He is a MIEEE, MACM, MISCI, LMCSI, MIETE, GMAIMA. He is a certified software quality auditor by MoCIT, Govt. of India.