Design and Development of Algorithms for Converting Parallel Regular Expressions to Deterministic Finite Automata

A Thesis

Submitted in fulfillment of the requirement

for the award of the degree of

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE AND ENGINEERING

Submitted by
Ajay Kumar
(Registration No. 950803001)

Computer Science & Engineering Department
Thapar University
Patiala - 147004
Punjab - India

August 2013

Dedicated to My Parents, wife Sunita Garhwal and My lovely daughter Aeshna(Tapu) Loura for their love, affection and motivation

CERTIFICATE

I hereby certify that the work which is being presented in this thesis entitled "***Design and Development of Algorithms for Converting Parallel Regular Expressions to Deterministic Finite Automata***" in fulfillment of the requirements of DOCTOR of PHILOSOPHY submitted in the Department of Computer Science and Engineering, Thapar university, Patiala is an authentic record of my own work carried out under the supervision of Dr. A. K. Verma and refers work of other researchers which are duly listed in reference section.

The matter embodied in this thesis has not been submitted for the award of any other degree of this or any other University.

A Kumar

(**Ajay Kumar**)

Registration No. 950803001

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge and belief.

Amit K. Verma 12/AUG/13.

(**Dr. A. K. Verma**)

Supervisor and Associate Professor

Computer Science and Engineering Department,

Thapar University, Patiala 147004

India

ACKNOWLEDGMENT

All praise due to *GOD*, the most beneficent, the most merciful, who bestowed upon me the courage, strength and patience to embark upon this work and carry it to completion.

I feel privilege to express my deep sense of gratitude and highest appreciation to my respected supervisor *Associate Professor, Dr. A. K. Verma, Computer Science and Engineering Department, Thapar University, Patiala*. I was providential to get a lifetime opportunity to work under his knowledgeable supervision. His association with this endeavor of mine will remain a beacon light to me through out my life. Thank you for making me a better person, guiding me and improving silliest and minutest mistakes. His presence was always there whenever I needed whether it was personal or professional life.

I express my gratitude to the Doctoral committee comprising *Dr. Maninder Singh* and *Dr. R.K. Sharma* for monitoring the progress and providing valuable suggestions for improvement of this Ph.D. work. I am very thankful to Dean of Research and Sponsored Projects *Dr. P.K. Bajpai*, Dean of students Affairs *Dr. Seema Bawa*, *Dr Inderveer Chana* PG coordinator and all faculty member for their valuable inputs and suggestions.

I would like to take this opportunity to express my gratitude towards *Dr. Brett Dominique Estrade, Perl Developer - Internal Systems, Cpanel Inc., Texas* for providing the useful guidance and material. I would like to thanks *Dr. Ram Jiwari* who enabled me not only to accomplish this huge task but also filled in me the courage to overcome the odds of life. His systematic approach and unconditional cooperation made me work hard. On a personal note, I would like to thank *Professor Vikram Singh*, CDLU for his valuable suggestions and supporting me at every moment of my life.

Here the time for the acknowledging friends comes. I was not sure to write this page as I didnt want to end up with a long list of ‘clich (which is, anyhow, exactly

what I am going to do!). The cooperation received through my friends *Amanpreet Sandhu, Dr. Pardeep Chauhan, Anoop Verma, Manwinder Kaur, Dr. Khuswant Singh, Dr. Gagan Kumar*.

I also thank those who could not find a separate name but helped me directly or indirectly in terms of reviewing and offering their valuable suggestions. I shall be thankful to all of my students of Thapar University and Mody Institute of Technology and Science who have studied Theory of Computation under my guidance and their constant response and valuable questions during the course inspire me to carry out this research work.

Above all it's my family who encouraged me to complete my PhD besides lots of odds and troughs in my life. I owe my personal reckoning of gratitude and benevolence to them for love, care, patience, inspiration, encouragement, motivation and constant driving force which has enabled me to complete my task. My parents have always supported me and to them I owe everything. My brother Arvind Loura and sister Anju have always supported and encouraged me to carry out this work.

I am profoundly grateful to my wife Sunita Garhwal who has always supported my aspirations with encouragement, proper guidance and taking care of me and my daughter at every turn of life. Even her cooperation cannot be expressed in words. I am greatly thankful to my sweet angel Aeshna (Tapu) Loura whose sweet smile and prayer to the god always inspire me for hardworking.

(Ajay Kumar)

ABSTRACT

Regular expressions are widely used in various applications such as *XML* schema languages, compiler designs, model checking, bio-informatics, virus detection systems, intrusion detection systems, finite state based testing, text processors and as a programming tool for multifarious scripting languages such as *PHP* and *PERL*, etc. A regular expression extended by intersection and shuffle operators do not increase the expressive power of the regular expression, but succinctness due to the inclusion of these additional operators are difficult to handle with the standard operators. In one form or another, the additional operator shuffle appears in different forms of computer science, namely process algebra, multipoint communication, *XML* schema *RELAX NG*, computational resource reduction, and concurrency of processes.

This dissertation focuses on the design of algorithm's for the conversion of regular expressions with additional operators to the finite automaton. Firstly, we develop an algorithm for the conversion of parallel regular expressions to non-deterministic finite automata using the follow-position of symbols presenting in the parallel regular expressions. It has also been proven that parallel regular expressions cannot be converted into deterministic finite automata directly. In the worst-case scenario, 2^{m+1} number of states are required for the construction of the non-deterministic finite automaton, where m represents the number of instances of alphabets in a parallel regular expression. In earlier existing approaches, the number of states of the non-deterministic finite automaton in the worst case is equal to $2^{2|r|-3C}$, where $|r|$ and C represent respectively the length and number of concatenation in a parallel regular expression.

Secondly, we explore and design a sound and complete algorithm for the conversion of semi-extended regular expressions to deterministic finite automata. Comparison demonstrates that the deterministic finite automaton generated using the

proposed algorithm is smaller than the earlier existing approaches in the literature. The proposed method is an extension of the direct conversion of regular expressions to deterministic finite automata.

Finally, we develop an algorithm for the metamorphosis of parallel regular expressions to the right linear grammar. The metamorphosis of parallel regular expressions to right linear grammar gives a prescript of reduction in concurrency. The novelty of the algorithm is the use of follow-position of symbols for the metamorphosis of parallel regular expressions to regular grammar.

Keywords: Regular expression, Parallel regular expression, Deterministic finite automata, Non-deterministic finite automata, Parallel finite automata.

PREFACE

This manuscript presents my research work done at the Computer Science and Engineering Department, Thapar University, Patiala, India from July 2008 to August 2013. This manuscript mainly focuses on design of three algorithms in the field of regular expressions with shuffle and intersection operators as follows:

1. Conversion of parallel regular expressions to non-deterministic finite automata and their state complexity issues.

2. An improved algorithm for the conversion of semi-extended regular expressions to deterministic finite automata.

3. An algorithm for metamorphosis of parallel regular expression to regular grammar.

The list of the main research papers accepted and communicated that I have authored or co-authored during my Ph.D. thesis are as follow:

RESEARCH PUBLICATIONS

1. Ajay Kumar, Anil Kumar Verma, Conversion of Parallel Regular Expressions to Nondeterministic Finite Automata using Partial Derivatives, *Accepted in Chiang Mai Journal of Science*, (SCI cited, Impact Factor=0.516).

2. Ajay Kumar, Anil Kumar Verma, A Novel Algorithm for the Conversion of Parallel Regular Expressions to Non-deterministic Finite Automata, Applied Mathematics & Information Science Vol. 8(1), PP.: 95-105, 2014, (SCI cited, Impact Factor=0.731).

3. Ajay Kumar, Anil Kumar Verma, A Novel Algorithm for the Conversion of Shuffle Regular Expressions to Non-deterministic Finite Automata, *Maejo International Journal of Science and Technology*, Vol. 7(3), PP.: 396-407, 2013, (SCI cited, Impact Factor=0.456).

TABLE OF CONTENTS

List of Abbreviations

DFA: Deterministic Finite Automaton

FA: Finite Automaton

NFA: Non-deterministic Finite Automaton

PFA: Parallel Finite Automaton

PRE: Parallel Regular Expression

RE: Regular Expression

RELAX NG: Regular Language description for XML Next Generation

SERE: Semi-extended Regular Expression

SRE: Shuffled Regular Expression

List of Figures

List of Figures

List of Tables

Chapter 1

INTRODUCTION

The purpose of this chapter is to introduce preliminary concepts, notations, motivations for the thesis, and the thesis outline.

1.1 MOTIVATION

Various models of computation, such as finite automata, push-down automata, linear bounded automata and the Turing machine can be defined mathematically. The Turing machine is equally as powerful as an actual computer. The simplest mathematical model is Finite Automata (*FAs*). The underlying principle of the finite automaton is that it consists of a finite number of states and can move among the states in a predictable way by accepting input signals. The language represented by the finite automaton is regular language. A regular language can be described by a regular expression, finite automaton and a regular grammar [37].

Regular expressions have been widely used since their appearance, and they play a significant role in myriad areas of computer science such as compiler design, protocol analysis, text processor, *XML* schema language, virus scanning system, operating system for specifying sharing of resources, bio-information, intrusion detection systems, natural language processing and as a programming tool for various scripting language such as *PHP*, *PERL* etc. [3-4, 9, 15, 17, 32, 47, 53, 56, 58, 63, 66].

Extended regular expressions consist of union, Kleene closure, concatenation, intersection, shuffle and counting operators. Use of these additional operators does not increase the expressive power of regular expressions, but the succinctness achieved by the addition of these operator is difficult to control using the union, Kleene

closure and concatenation operators. An extended regular expression can be rewritten into an equivalent regular expression, but the equivalent regular expression is usually larger and more complicated. Therefore, extended regular expressions are preferred to regular expressions. Inclusion of these additional operators means that the patterns are represented more elegantly and are easier to understand. Regular expressions with subtraction operators can be useful in programming languages, such as for describing an identifier using certain standard patterns of regular expressions minus a set of keywords. Regular expression does not provide an adequate way of depicting this. Semi-extended regular expressions consists of concatenation, union, intersection and Kleene closure operators.

Parallel Regular Expressions (*PREs*) [5] consist of shuffle, union, concatenation and Kleene closure operators. *PREs* are useful in different areas of computer science namely multipoint communications, natural language processing, pattern matching, *XML* schema *RELAX NG*, concurrency of processes, system verification etc. [35, 67, 74, 76].

The concept of shuffle operators in regular expressions was introduced by Garg and Ragunath [70], they specified that the shuffle operators are useful in the modelling of interleaving processes and that there is a close relationship between petri nets and shuffle languages.

Carberry [61] proposed various techniques for plan recognition. In plan recognition, an agent's goal is identified on the basis of the agent's actions [10, 61]. Hogberg [38] proposed a model-based approach to represent a multi-agent scenario in which shuffle operators are used in context-free languages.

Iwama [41] used shuffle operators in multi-point communication, where all stations share a single bus. The decomposition of a string x into $y_1, y_2, ..., y_n$ is such that each decomposition belongs to a predetermined language, $L_1, L_2, ..., L_n$. Conditions under which such a decomposition is unique are extremely important in multi-point

communication.

Estrade *et al.* [6] investigated various applications of the parallel regular expressions in the field of modelling of multitasking and multiuser operating systems. Parallel regular expressions can be used for describing a number of competing processes, with each symbol representing a critical section. For a single CPU, the equivalent regular expression corresponding to a parallel regular expression describes how these processes can be scheduled. Parallel regular expressions are also useful in the modelling of concurrent systems. In the event that the number of processors is reduced, partial serialization can be useful [5].

The design of efficient algorithms for the conversion of regular expressions with additional operators to deterministic and non-deterministic automata has become crucial, as these are widely used in various fields of computer science.

1.2 TERMINOLOGY

In this section, we describe some basic definitions and notations that are used throughout the remaining part of the thesis.

Σ denotes an alphabet representing a finite non-empty set of symbols. Σ^* is the free monoid generated by concatenation on Σ including the null string [37]. A string or word $w = a_1 a_2 ... a_n$ is a finite sequence of symbols taken from Σ, having length n. Length of a string w is denoted by $|w|$. Empty string (represented by ε) does not contain any symbol. A language L is a subset of Σ^*. Null language (denoted by $\emptyset$) does not consist of any string.

DEFINITION 1.2.1. Language L [37] 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. $L(r_1 r_2) \leftarrow L(r_1)L(r_2)$

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 1.2.2. *DFA* [37] 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 $Q \times \Sigma \rightarrow Q$.

5. Set $F \subseteq Q$ is the set of final states.

Figure 1.2.1 delineates the *DFA* for the language $L = (a|b)^*ba$. Throughout the thesis, the final state of the finite automaton is represented by green color circle or by double circle.

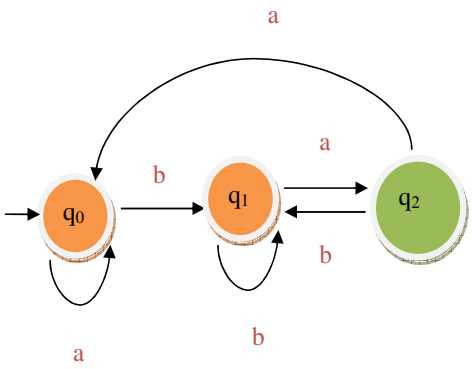


Figure 1.2.1: *DFA* for $L = (a|b)^*ba$

DEFINITION 1.2.3. *NFA* [37] 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.

Figure 1.2.2 delineates the *NFA* for the language $L = (a|b)^*ab$.

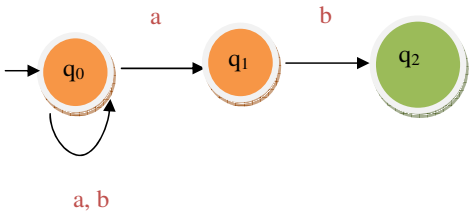


Figure 1.2.2: *NFA* for $L = (a|b)^*ab$

DEFINITION 1.2.4. Shuffle operator is denoted by &. Shuffling [50] 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 1.2.1. Consider $w_1 = abc$ and $w_2 = xy$, then

$$w_1\&w_2 = \{abcxy, xyabc, axbcy, axybc, axbyc, abxyc, xabcy, xaybc, xabyc, abxcy\}$$

DEFINITION 1.2.5. 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 1.2.6. Syntax tree of *RE r* is a binary tree in which every node has at most two children and the following conditions holds:

1. $a \in \Sigma$, # 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 *RE*.

DEFINITION 1.2.7. Language L [6] 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. $L(r_1 r_2) \leftarrow L(r_1)L(r_2)$
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 1.2.8. *PFA* [69] 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 $Q \times (\Sigma \cup \lambda) \rightarrow 2^Q$.

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

Figure 1.2.3 delineates the *PFA* for the language $L = 0\&(01)$.

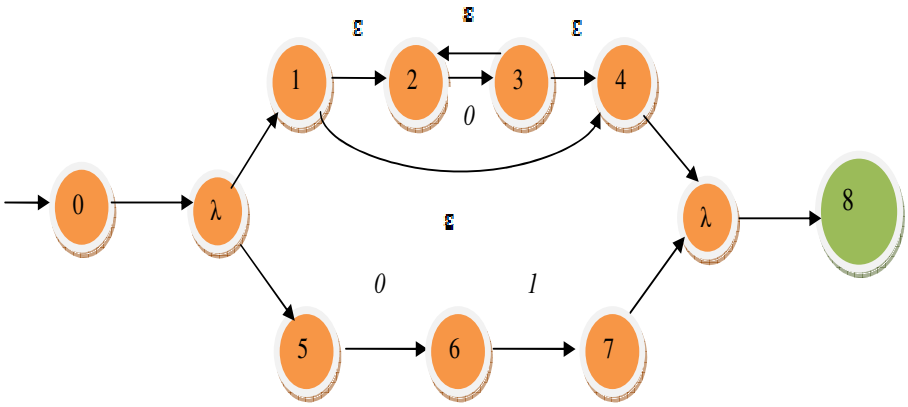


Figure 1.2.3: *PFA* for $L = 0\&(01)$

1.3 Thesis Outline

This thesis is devoted to the design of algorithms for the conversion of *PREs* to *NFAs*. The results obtained are discussed in chapter 3 to chapter 7. A chapter-wise summary of the thesis is as follows:

The **first chapter** is introductory and serves the purpose of developing the preliminary concepts, including definitions and basics, keeping in view the pre-requisites for the subsequent chapters.

In chaper 2, we discuss various methodologies that exists in the literature for the conversion of *RE* to *DFA* or *NFA*. This chapter also contains the literature survey of *PRE* and their conversion to *NFA*. In last section, gaps in the area of *PRE* are explored.

In **chapter 3**, an algorithm is designed for the conversion of shuffled regular expression (*SRE*) to *NFA*. We have proved that a *PRE* cannot be converted into *DFA* directly. *NFA* generated using the proposed approach requires 2^{m+s+1} number of states in the worst-case scenario, which is a significant improvement over the other existing approaches in the literature. It has been proved that if $\forall(r_i \& r_j) \in r \wedge ((symbol(r_i) \cap symbol(r_j)) \neq \emptyset)$, then it is implausible to generate deterministic finite automaton directly from the parallel regular expression.

In **chapter 4**, an algorithm is designed for the metamorphosis of *PREs* to *NFAs* without using intermediate steps. Estrade *et al.* [6] investigated the *PREs* and proposed an algorithm for the conversion of a *PRE* to *NFA* using *PFA* as an intermediate step. The proposed algorithm is an extension of the direct conversion method used for converting a *RE* to *DFA*. For a given *PRE* r, let m be the number of symbols that occur in r and let C denote the number of concatenation operators in r, then in the worst case, 2^{m+1} states are required for the construction of the *NFA* using the proposed algorithm. In the earlier existing approaches, the number of states of the *NFA* in the worst case is equal to $2^{2|r|-3C}$.

In **chapter 5**, the worst case state complexity of the ε-free *NFA* is analyzed using the partial derivative automata. The study of derivatives of *RE* were introduced by Brzozowski [33], and by applying this algorithm, a *RE* can be converted into *DFA*. Later on, Antimirov [68] proposed the concept of partial derivatives for the conversion of *RE* to *NFA*. It has been proven that the worst case *NFA* (i.e. *NFA* having $m+1$ states) occurs using partial derivatives, if m instances of symbols are connected by $(m-1)$ concatenation operators and no other operators are used. Various properties of the *PRE* have been explored, as well as how partial derivatives are useful for the conversion of *PRE* to *NFA*.

In **chapter 6**, an algorithm is designed for the conversion of *PRE* to right linear grammar. The proposed algorithm uses the concept of follow-position of the

symbols. The metamorphosis of *PRE* to regular grammar gives a prescript of reduction in concurrency, which can be credited as a novel algorithm. The numerical implementation of the algorithm is outlined, and some specific numerical examples are given to illustrate the proposed conversion. Furthermore, the grammar procured from the proposed algorithm can be easily converted into a minimized *DFA*.

In **chapter 7**, an algorithm is designed based on the follow-positions of symbols present in the *SERE* for the conversion of *SERE* to *DFA*. Comparison demonstrates that the *DFA* generated using the proposed algorithm is smaller than the earlier existing approaches in the literature. The proposed algorithm is an extension of the direct conversion method used for converting the regular expressions into deterministic finite automata whereas Yamamoto algorithm [31] is based on the Glushkov automata [72]. Finally, to expound this approach, we endow some numerical examples.

Finally, conclusions are presented, followed by the future work and cessation with references, including papers, books and websites.

Chapter 2

PRIOR WORK

Over the last six decades, extensive research has been conducted on the conversion of regular expressions to deterministic and non-deterministic finite automata. In this chapter, we will first review some of the techniques for the conversion of regular expressions to finite automata. We will also review the literature on the conversion of regular expressions with additional operator shuffle and intersection to the finite automata.

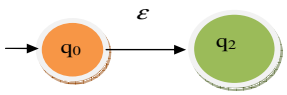
2.1 FROM REGULAR EXPRESSIONS TO NFAS

Kleene [62] described the simplest model of computation in 1956, since known as deterministic finite automaton (*DFA*). He proposed a specification language known as regular language and used notation nerve nets in place of *DFAs*. He also proposed the Kleene theorem, which states that every regular expression (*RE*) can be represented by a non-deterministic finite automaton (*NFA*).

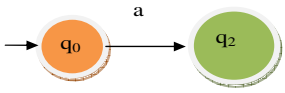
Thompson [43] proposed a classical approach, by means of which a regular expression can be converted into an ε-*NFA* in linear time. Thompson's construction is still used in many applications because it is easy to implement. Given *RE r* having length *n*, ε-*NFA* can be generated using Thompson's construction requires atmost *2n* states and *4n* transitions. Figure 2.1.1 delineates various rules for Thompson's construction.

Sippu and Soisalon-Soininen [65] proposed an approach whereby a smaller $\varepsilon - NFA$ can be constructed than is possible when using the Thompson's automata. Figure 2.1.2 delineates various rules for union, concatenation and Kleene closure using the Sippu and Soisalon-Soininen construction.

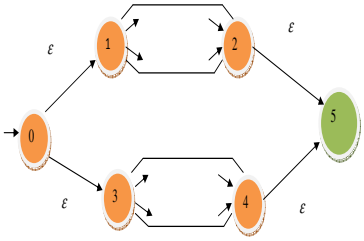
(a) *RE*=∅



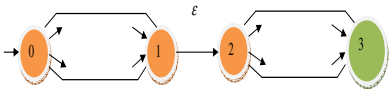
(b) *RE*=ε



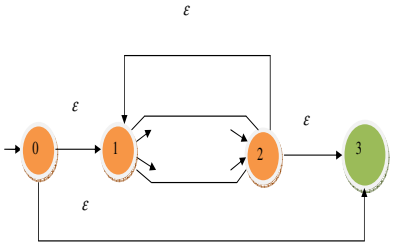
(c) *RE*=*a*



(d) Union operation

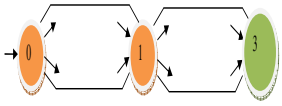


(e) Concatenation operation

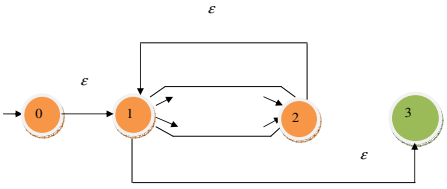


(f) Kleene closure operation

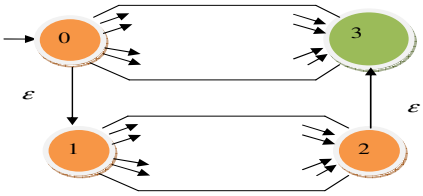
Figure 2.1.1: Thompson's Construction [43]



(a) Concatenation operation

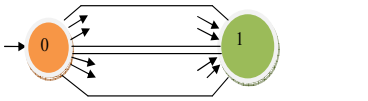


(b) Kleene closure operation

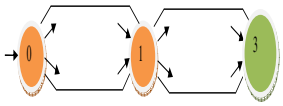


(c) Union operation

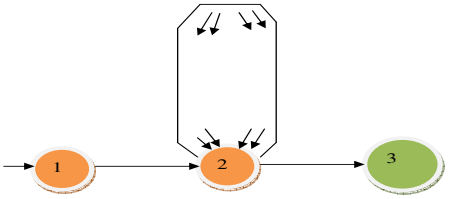
Figure 2.1.2: Sippu and Soisalon-Soininen Construction [65]



(a) Union operation



(b) Concatenation operation



(c) Kleene closure operation

Figure 2.1.3: Ilie and Yu Construction [44]

Ilie and Yu [44] introduced the concept of follow automata. This is based on the equivalence relation of the set of positions of symbols occurring in a regular expression. The *NFA* obtained using follow automata consist of $(m+1)$ states and (m^2+m) transitions in the worst-case scenario where m be the number of symbols that occur in *RE* r. Follow automata can be constructed by merging the equivalent states of the Glushkov automata. Ilie and Yu [45] gave a method which can convert a *RE* into *NFA* whose size is less than or equal to the size of the *NFA* obtained using the position, partial derivative and follow automata.

The concept of $\varepsilon-$free *NFA* is introduced as position automata, proposed independently by McNaughtonYamada [59] and Glushkov [72]. The size of the *NFA* obtained using the position automata lies between the linear and the quadratic. Using Bruggemann-Klein's [1] approach, the position automata can be obtained in quadratic time. In positional automata, each occurrence of the symbol from an alphabet is assigned a position and each position is considered as a state. A limitation of the position automata is that the number of transitions can be equal to m^2+m in the worst-case scenario. In position automata, *first*, *last* and *follow*(i) are determined using the following rules:

$$first = \{j | a_j \alpha \in L(r)\}$$
$$last = \{j | \alpha a_j \in L(r)\}$$
$$follow(j) = \{i | \alpha a_j a_i \beta \in L(r)\}$$

In position automata, each state represents the occurrence of a symbol from the symbols in r. Consider $r = ab(a|b)^*$. Figure 2.1.4 represents the position automata equivalent to $L(r)$. State 1 represents the occurrence of symbol a, state 2 represents the occurrence of symbol b and so on.

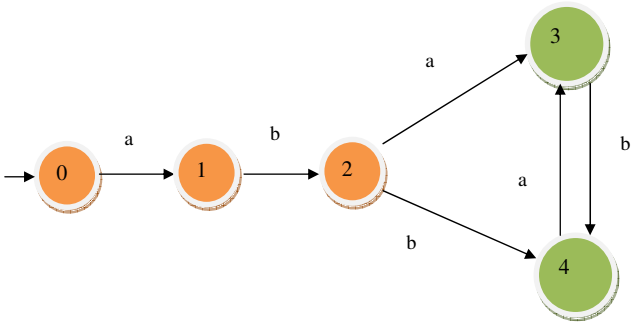


Figure 2.1.4: Position automaton for *RE* $r = ab(a|b)^*$

Brzozowski's [33] proposed the concept of word derivatives for the conversion of *REs* to *DFAs*. Antimirov [68] used the concept of partial derivatives for the construction of *NFAs* from a regular expression having the time complexity $O(n^5)$. Champarnaud and Ziadi [39] improved the time complexity of Antimirov's partial derivatives with quadratic time complexity. They proved that the partial derivative automata are a quotient of position automata.

Consider a language L over an alphabet $\{a, b\}$ denoted by *RE* r. Then, according to Antimirov [68] following are the properties of partial derivatives:

$$\partial_a(\varepsilon) = \emptyset$$
$$\partial_a(\emptyset) = \emptyset$$
$$\partial_a(a) = \varepsilon$$

$$\partial_a(b) = \emptyset$$
$$\partial_a(x \cup y) = \partial_a(x) \cup \partial_a(y)$$
$$\partial_a(x^*) = \partial_a(x)(x^*)$$
If $\varepsilon \in L(x)$ then
$$\partial_a(xy) = (\partial_a(x)y) \cup \partial_a(y)$$
If $\varepsilon \notin L(x)$ then
$$\partial_a(xy) = (\partial_a(x)y)$$

Berry and Sethi [20] used Brzozowski's results [33] and improved the McNaughton and Yamada's algorithm [59] for the metamorphosis of *REs* into *NFAs*. Klein [1] converts the *RE* into star normal form and proposed a two-pass algorithm for computing McNaughton and Yamada's [59] *NFA* using the same kind of mechanism used by Berry and Sethi [20]. Ziadi and Champarnaud [16] proposed an optimal algorithm for converting a regular expression into Glushkov automata using star normal form. Broda *et al.* [60] studied the relation between partial derivative automaton and Glushkov automaton.

Garcia *et al.* [57] proposed a methodology using which a smaller *NFA* can be constructed. They propose an algorithm, using which the states of the c-continuation automaton are merged if they satisfy the follow automata property keeping the c-continuation of the merged states. This give each merged states having several expressions representing the same language. Then, the states which are having a common regular expression are also merged. The size of the obtained *NFA* is upper bound of the size of smallest automata obtained from the existing approaches in the literature [57].

Giammarresi *et al.* [11] characterized the Thompson digraphs and proposed an algorithm using which a regular expression can be generated from a Thompson machine having size linear in terms of the total number of states and transitions. Castiglione *et al.* [21] proposed the concept of non-deterministic Moore automaton

and showed that it has the same computational power as minimal deterministic Moore automaton. Sakuma *et al.* [78] worked on the strategy of regular expression used in Perl and translated a regular expression into transducers. Several researchers investigated different types of open problems and explored various applications in the area of regular expressions [40, 71, 79-80].

2.2 FROM EXTENDED REGULAR EXPRESSION TO NFA

Garg [69] proposed the concept of parallel regular expressions (*PREs*) for the modeling and analysis of concurrent systems. It is proved that *PREs* can represent the similar language as petri nets [70].

Stotts and Pugh [56] introduced that Parallel Finite Automata (*PFAs*) can be used for describing the finite automata with interleaving operators. They specify that the *PFAs* can combine the modelling capabilities of petri nets without admitting the possibility of an infinite state space. An algorithm was designed for converting parallel regular expressions to *PFAs* using composition rules. *PFAs* can be used in the modelling of hypertext and nonlinear interactive networks.

Mishna *et al.* [52], worked on shuffle product, shuffle closure and shuffle grammar and they gave explicit generating function consequences. Pardeep and Murthy [7] designed a constant time algorithm for determining whether a string can be generated from the shuffling of two strings. Hovland [12, 13] proposed a membership algorithm for *RE* consisting of union, concatenation, Kleene closure and unordered concatenation.

Estrade *et al.* [6] designed an algorithm for the conversion of *PREs* to *DFAs* using the intermediate step of *PFAs* and *NFAs*. They designed a Formal Language Toolkit (FLaT) [28] for the conversion of *PREs* to *DFAs*. The conversion of *PREs* to *DFAs* serializes the parallelism and concurrency. *PRE* to *PFA* conversion can be

achieved by using the modified Thompson's construction [5]. Conversion of *PFAs* to *NFAs* requires the elimination of λ-transition and this can be done by enumerating all states explicitly. *NFA* to *DFA* conversion can be achieved by the well-known subset construction technique. Estrade *et al.* [6] investigated different application of parallel regular expressions in the field of computation resource reduction, modelling a multi-tasking and multiuser operating system. They also explored some potential areas for future research directions.

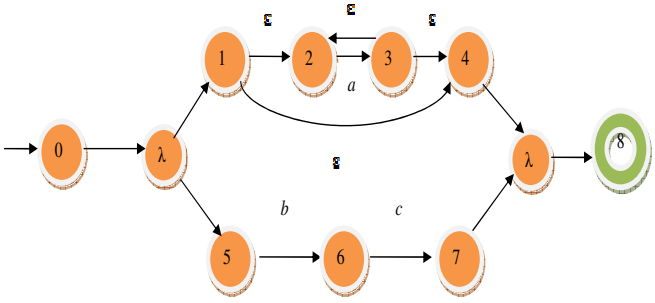


Figure 2.2.1: *PFA* for *PRE* $r = a^*\&bc$ using Estrade *et al.* methodology [6]

Table 2.2.1: Execution sequence of $w = baca$ for *PFA*

Active node	**Input**	**Action**
0	λ	Control goes to $\{1,2,4,5\}$
$\{1,2,4,5\}$	b	Control goes to $\{1,2,4,6\}$
$\{1,2,4,6\}$	a	Control goes to $\{2,3,4,6\}$
$\{2,3,4,6\}$	c	Control goes to $\{2,3,4,7\}$
$\{2,3,4,7\}$	a	Control goes to $\{2,3,4,7\}$
$\{2,3,4,7\}$	λ	Control goes to $\{8\}$

Berglund [48] studied the complexity of the membership problem involved in the shuffled language. Berstel *et al.* [34] studied the different kinds of language classified

based on the shuffle operations and proved partial results for the star-free commutative language. Micciancio [14] proved the decidability of valid equations and inequations between regular expressions containing intersection and shuffle operators. Bjorklund and Bojanczyk [23] worked on shuffled expressions and words with nested data. They showed that the emptiness is decidable in case of unordered data and undecidable with ordered data.

Semi Extended Regular Expressions (*SEREs*) consisting of concatenation, union, intersection and Kleene closure operators. In many applications, the words represented by the patterns are long and cannot be easily depicted by the *REs*. These patterns can be easily depicted by using *SEREs*.

Yamamoto [31] proposed an algorithm for the metamorphosis of *SEREs* into Non-deterministic Finite Automata (*NFAs*). This algorithm is founded on the concept of Glushkov Automata [55]. Yamamoto [30] designed an algorithm for the membership of *SEREs* using its metamorphosis to partially input-synchronized alternating automata.

Kupferman and Zuhovitzky [54] designed the improved algorithm for the membership of *SEREs* on the basis of alternating automata with synchronized universality and negation. It is arduous for the conversion of a *SERE* into alternating automaton whenever sub-expression $(r_i \cap r_j)r_k$ occurs in the *SERE*. Problem occurs because sub-expression $(r_i \cap r_j)r_k$ is not equal to $(r_i r_k) \cap (r_j r_k)$ whereas $(r_i \cup r_j)r_k$ is equal to $(r_i r_k) \cup (r_j r_k)$.

A number of membership algorithms exist for checking whether string w can be produced from the *SERE* or not. Useful membership algorithms for *SEREs* have been provided by Yamamoto [30] and extensively improved by Kuperferman and Zuhovitzky [54], Rosu [22], Peterson [29].

In recent years, the succinctness of *REs* with additional operators like shuffle, intersection [24, 25, 42, 74, 75] has been widely studied and reasserted that double

exponential size is needed for the construction of *RE* from *RE* with intersection operators.

2.3 GAPS IN THE LITERATURE

Shuffle decomposition for a regular language L is the decomposition of L into two regular languages L_1 and L_2 such that $L = L_1 \& L_2$. Campeanu *et al.* [8] proved that for certain sub-classes (like commutative regular languages) of regular language, shuffle decomposition is decidable and undecidable for context-free languages.

Ito showed that if one of the component languages is restricted to be finite then, shuffle decomposition is decidable [51]. Domaratzki *et al.* [49] considered the concepts of decomposition of unary regular language along with a regular set of trajectories to obtain the decidability of the unary regular language. Biegler *et al.* [18] proved that the shuffle decomposition of finite languages is not unique but shuffle decomposition is unique for words and arbitrary sets.

From the above stated work, it has been shown that knowing exactly which finite sets have a unique decomposition is a still an open problem as is the status of decidablity.

Estrade *et al.* [6] suggest that a serialized form cannot be easily recognized from its parallel regular expression. Further, unshuffling of *PREs* (i.e. converting a serial regular expression back into source language *PRE*) is still an problem. The open issues are to convert *PREs* into invertible forms and the ability to correctly identify the source languages. As a result, it leads to a high complexity of the problem, but also offers a greater possibility to find the nested level of concurrency.

According to Estrade *et al.* [6], an algorithm can be designed for the direct conversion of *PREs* to *DFAs*, which can be useful to study the properties of shuffle operators and useful in the field of concurrency.

2.4 Objectives of the Research Work

The main objective of this research work is the design of algorithms for the conversion of *PREs* to *DFAs*. Based upon the literature survey and the gaps, following research objectives are outlined for the proposed work:

1. To explore various algorithms in the field of Parallel Regular Expressions (*PREs*) and Parallel Finite Automata (*PFAs*).
2. To explore various techniques / technologies for the conversion of *PREs* to finite automata.
3. To propose an efficient algorithm to convert *PREs* to *DFAs*.
4. To Verify and Validate the proposed algorithm.

Based upon the above objectives the outcomes of this research is the design of an algorithm for the conversion of *PRE* to *NFA*.

(a) Estrade *et al.* approach [6] **(b)** Proposed Approach

Figure 2.4.1: Comparison between Estrade *et al.* approach [6] and proposed approach

In addition to the above mentioned objectives following algorithms are also designed:

1. Conversion of the Semi-extended Regular Expressions (*SEREs*) to *DFAs*.
2. Conversion of *PREs* to right linear grammars.

Chapter 3

A NOVEL ALGORITHM FOR THE CONVERSION OF SHUFFLED REGULAR EXPRESSIONS TO NON-DETERMINISTIC FINITE AUTOMATA

3.1 INTRODUCTION

Shuffled Regular Expression (*SRE*) is a confined form of Parallel Regular Expression (*PRE*) in which shuffle operators explicitly occur between Regular Expressions (*REs*). Conversion of *SRE* to *RE* or Deterministic Finite Automaton (*DFA*) has been studied widely in the area of concurrency.

SREs can be used for depicting the number of processes competing for the critical section [5]. Each critical section of a process can be described by a symbol. For example, critical sections of the three processes could be represented by a, $(ab)^*$ and b^*. For a single CPU, *SRE* describes the probabilistic way of scheduling. Equivalent *RE* commensurate to the *SRE* renders the way for tasks to be completed. In case of resource reduction in the modelling of concurrent systems, partial serialization can be useful [6]. In this chapter, an algorithm is proposed for the metamorphosis of *SREs* to ε*-free* Non-deterministic Finite Automata (*NFAs*) without using intermediate steps.

Various methodologies exist in the literature for the metamorphosis of *REs* to *NFAs* named as Thompson's construction [43], partial derivatives [68], and follow automata [44]. Estrade *et al.* [6] worked on the conversion of Parallel Regular Expression (*PREs*) to *NFAs* using Parallel Finite Automata (*PFAs*) as an intermediate state. *PRE* to *PFA* conversion can be achieved by using the modified Thompson's construction. *PFA* with n states can be converted into a *NFA* require 2^n states in the worst-case scenario [6]. However, *RE* can be converted into *DFA*

[1]Results of this chapter are published in *Maejo International Journal of Science and Technology*, Vol. 7(3), 396-407, 2013.

directly with fewer states than the trivial Thompson's construction. Motivation for developing an efficient algorithm, with lesser number of states stems as there is a scope for conversion of *PRE* to *NFA* by extending the direct conversion method [3, 20].

3.2 PROPOSED APPROACH

In direct conversion [3, 20] of *REs* to *DFAs*, a syntax tree is constructed for the augmented *RE*. *firstpos*, *lastpos* and *nullable* are determined using the rules given in Table 3.2.1. The additional rules for shuffle operator are described along with the rules for the direct conversion of *RE* to *DFA* [20]. *followpos* are determined using the rules given in Table 3.2.2. *firstpos*, *lastpos*, *followpos* and *nullable* are determined for the nodes of the syntax tree, and using these *followpos*, an equivalent *DFA* can be generated.

Table 3.2.1: Rules for *firstpos*(r), *lastpos*(r) and *nullable*

RE	***firstpos*(r)**	***lastpos*(r)**	***Nullable***
$r = \emptyset$	$\emptyset$	$\emptyset$	*false*
$r = \varepsilon$	$\emptyset$	$\emptyset$	*true*
$r = a$	$position(a)$	$position(a)$	*false*
$r = r_i \cup r_j$	$firstpos(r_i) \cup firstpos(r_j)$	$lastpos(r_i) \cup lastpos(r_j)$	$nullable(r_i) \vee nullable(r_j)$
$r = r_i r_j$	**If** $(\varepsilon \in L(r_i))$ $firstpos(r_i) \cup firstpos(r_j)$ **Else** $firstpos(r_i)$	**If** $(\varepsilon \in L(r_j))$ $lastpos(r_i) \cup lastpos(r_j)$ **Else** $lastpos(r_j)$	$nullable(r_i) \wedge nullable(r_j)$
$r = r_i^*$	$firstpos(r_i)$	$lastpos(r_i)$	*true*
$r = r_i \& r_j$	$firstpos(r_i) \cup firstpos(r_j)$	$lastpos(r_i) \cup lastpos(r_j)$	$nullable(r_i) \wedge nullable(r_j)$

SRE r with *s* shuffle operators can be dispensed into *s+1* sub-regular expressions $r_1, r_2, ..., r_{s+1}$ such that:

1. $r = r_1 \& r_2 \& \ldots \& r_{s+1}$

Table 3.2.2: Rules for *followpos* [3]

RE	***followpos***
$r_i r_j \in r$	For $p \in lastpos(r_i)$ $followpos(p) \leftarrow followpos(p) \cup firstpos(r_j)$
$r_i^* \in r$	For $p \in lastpos(r_i)$ $followpos(p) \leftarrow followpos(p) \cup firstpos(r_i)$

2. $r_1, r_2, ..., r_{s+1}$ are the *REs*.

An augmented *SRE* can be obtained by adding # before each shuffle operator and at the end of *SRE* r. The first step is to construct a syntax tree (similar as shown in Figure 3.2.1) for the augmented *SRE*, such that symbols and #'s appear at leaf nodes, and operators appear as the internal nodes. For all leaf nodes, except a node labeled with #, *followpos* are calculated. *followpos*(#) is taken as null set.

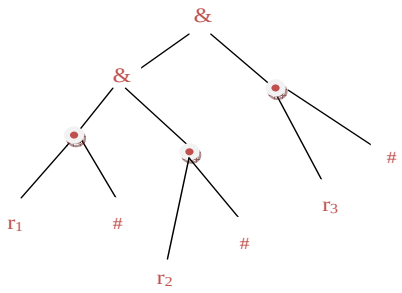


Figure 3.2.1: Syntax tree for the augmented *PRE* $r = (r_1\#\&r_2\#\&r_3\#)$

followpos for each sub-regular expression is calculated separately. Consider, *SRE* $r = r_i \& r_j$ having a single shuffle operator. *firstpos*, *lastpos* and *followpos* of r_i and r_j are calculated separately. The starting state of the *NFA* is the union of $firstpos(r_i)$ and $firstpos(r_j)$. *lastpos* of the root node is determined by taking the union of all positions labeled with #. State q_i is a final state if $lastpos(root) \subseteq q_i$. Using the definition of the shuffle, the resulting *NFA* must include all strings generated from the shuffle of the strings generated from r_i with the strings generated from r_j. The final state of the

NFA should accept all strings generated from $r_i\&r_j$. At any instant of time during the processing of string w, consider q_i is the current state. On reading symbol a, from the sub-regular expressions r_i and r_j, the following two cases can occur after reading symbol a as depicted in Figure 3.2.2.

1. Reading of symbol a, from r_i:

   $q_j \leftarrow followpos((position(r_i)\ labeled\ with\ a)\ in\ q_i) \cup (positions(r_j)\ in\ q_i)$

2. Reading of symbol a, from r_j:

   $q_k \leftarrow followpos((position(r_j)\ labeled\ with\ a)\ in\ q_i) \cup (positions(r_i)\ in\ q_i)$

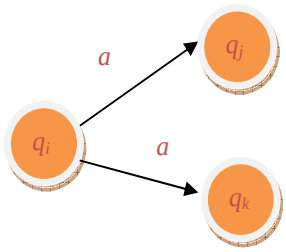


Figure 3.2.2: Reading of symbol a from $(r_i\&r_j)$

Similarly, there is a possibility of reading symbol a, on states q_j and q_k from sub-expressions r_i and r_j respectively. Hence, if $((symbols(r_i) \wedge symbols(r_j)) \neq \emptyset) \wedge (r_i\&r_j \in r)$, then an *NFA* is generated from r using the proposed approach. Repeating this procedure, all possible shuffled strings can be generated. The final state of the *NFA* will accept all strings which are obtained by $r_i\&r_j$. In general, *SRE* r consists of *(s+1)* sub-regular expressions $(r_1, r_2, r_3, ..., r_{s+1})$, such that $r = r_1\&r_2\&r_3\&...\&r_{s+1}$. At any instant of time during the processing of a string w, consider q_i is the current state. Symbol a can be taken from sub-regular expressions r_x, r_y *and* r_z of the state q_i. Following cases can occur after reading symbol a:

1. Reading of symbol a from r_x:

   $q_j \leftarrow followpos((position(r_x)\ labeled\ with\ a)\ in\ q_i) \cup (q_i - positions(r_x))$

2. Reading of symbol a from r_y:

   $q_k \leftarrow followpos((position(r_y)\ labeled\ with\ a)\ in\ q_i) \cup (q_i - positions(r_y))$

3. Reading of symbol a from r_z:

$$q_l \leftarrow followpos((position(r_z)\ labeled\ with\ a)\ in\ q_i) \cup (q_i - positions(r_z))$$

Further, reading of symbol a, from the sub-expressions r_y and r_z on state q_j is possible. Similarly, there is a possibility of reading symbol a, on states q_k and q_l. Repeating this procedure, all possible shuffled strings can be generated from the sub-regular expressions $r_1, r_2, ..., r_{s+1}$. The final state will accept all strings of the *SRE* r. If a common symbol appears in different sub-regular expression of r, involved in shuffling, then *DFA* cannot be generated directly from *SRE* r.

Notation used in the algorithm: The following notation are used in the algorithm:

a: It indicates any symbol from Σ.
***c_1*:** It represent *left* child of a node or single child of Kleene closure.
***c_2*:** It represent *right* child of a node.
***Position*(n):**It depicts the positions of the symbols from *RE* embarked at node n.

Using Klein's [1] methodology; *firstpos*, *lastpos*, and *followpos* are determined for all nodes except for shuffle operator. The rules for determination of *firstpos*, *lastpos*, *nullable* and *followpos* are depicted in Table 3.2.1 and Table 3.2.2. The same is represented in the procedure *Calculate_Follow*.

Algorithm 3.2.1: *SRE_NFA*(p, NFA)

Input: *SRE p*

Output: $\varepsilon-$free *NFA M* such that *L(M)=L(p)*

1 Generate augmented *SRE p*
2 Construct a syntax tree for augmented *SRE p*. ▷ Syntax tree construction
3 *Calculate_Follow*(p). ▷ Call to Procedure *Calculate_follow*
4 *Create_NFA*(*firstpos*,*lastpos*, *followpos*) ▷ Call to Procedure *Create_NFA*

Algorithm 3.2.2: *Calculate_Follow(p)*

Input: *Syntax tree for the SRE p.*
Output: *firstpos, lastpos and followpos.*

```
1  for (n ∈ postorder(syntax tree)) do
2      ▷ n is the current node during postorder traversal of a syntax tree
3      if ((n = a) ∨ (n = #)) then
4          firstpos(n) ← position(n)                                  ▷ Leaf node
5          lastpos(n) ← position(n)
6      end
7      if (n = |) then
8          firstpos(n) ← firstpos(c1) ∪ firstpos(c2)                  ▷ Union node
9          lastpos(n) ← lastpos(c1) ∪ lastpos(c2)
10     end
11     if (n = .) then
12         firstpos(n) ← firstpos(c1)
13         lastpos(n) ← lastpos(c2)
14         if (c1 = ∗) then
15                                          ▷ Nullable at left child is true
16             firstpos(n) ← firstpos(n) ∪ firstpos(c2)
17         end
18         if (c2 = ∗) then
19                                          ▷ Nullable at right child is true
20             lastpos(n) ← lastpos(n) ∪ lastpos(c1)
21         end
22         for (i ∈ lastpos(c1)) do
23             followpos(i) ← followpos(i) ∪ firstpos(c2)
24         end
25     end
26     if (n = ∗) then
27                                                          ▷ Kleene closure
28         firstpos(n) ← firstpos(c1)
29         lastpos(n) ← lastpos(c1)
30         for (i ∈ lastpos(n)) do
31             followpos(i) ← followpos(i) ∪ firstpos(n)
32         end
33     end
34     if (n = &) then
35                                                          ▷ Shuffle operator
36         firstpos(n) ← firstpos(c1) ∪ firstpos(c2)
37         lastpos(n) ← lastpos(c1) ∪ lastpos(c2)
38     end
39 end
```

Algorithm 3.2.3: *Create_NFA* (*firstpos*, *followpos*, *lastpos*)

Input: *firstpos, lastpos and followpos of SRE p.*
Output: *NFA* equivalent to *SRE p*.

```
1  q_0 ← firstpos(root)
2  s ← number of shuffle operator in p
3  Q ← q_0 and make q_0 as unmarked.
4  while ((q ∈ Q) ∧ (q is unmarked)) do
5      Choose an unmarked state q and mark it.
6      Prevlast ← 0
7          ▷ Prevlast indicates the last position of the sub-expression recently processed
8      i ← 1
9      while (i ⩽ s + 1) do
10                                          ▷ Reading of a on the current sub-expression
11         currentlast ← i^th value of lastpos(root)
12         T ← ∅
13         for (∀pos ∈ q) do
14             if (Prevlast < pos ⩽ currentlast) then
15                 T ← T ∪ pos
16             end
17         end
18         N = q − T
19         for (∀a ∈ Σ) do
20             Newstate ← ∅
21             for (∀t ∈ T) do
22                 if (symbol(t) = a) then
23                     Newstate ← Newstate ∪ followpos(t)
24                 end
25             end
26         end
27         if (Newstate ≠ ∅) then
28             Newstate ← Newstate ∪ N
29             Tran[q, a] ← Newstate
30             if (Newstate ∉ Q) then
31                 Q ← Q ∪ Newstate
32                 Make Newstate as unmarked state.
33             end
34         end
35         Prevlast ← currentlast
36         i ← i + 1
37     end
38 end
39 q_0 = {q|q = firstpos(root)}                                  ▷ Starting state
40 F = {q_f|(q_f ∈ Q) ∧ (lastpos(root) ⊆ q_f)}                   ▷ Final state
```

The reading of a symbol on a state is processed by two steps. In the first step, we find the *followpos* of the current symbol from a sub-regular expression. In the second step, we add the positions of other sub-regular expressions involved in a shuffling of the current state to the result of the first step. This process is repeated for each $q \in Q$ and each $a \in \Sigma$. Initial state and final state are determined using *firstpos* and *lastpos* of the root node respectively. Complete procedure is specified in Algorithm 3.2.3.

The complete procedure for the conversion of *SRE* to *NFA* is expounded by the following numerical example.

Example 3.2.1. *Given* $r = (a(a|b)^*)\&(a^*b^*)$

Augmented $r = (a_1(a_2|b_3)^*)\#_4\&(a_5^*b_6^*)\#_7$

Figure 3.2.3 delineates the syntax tree for the augmented *SRE*. The *firstpos* of a node is depicted on the left side of the node in the blue color, and the *lastpos* of a node is depicted on the right side of the node in the green color.

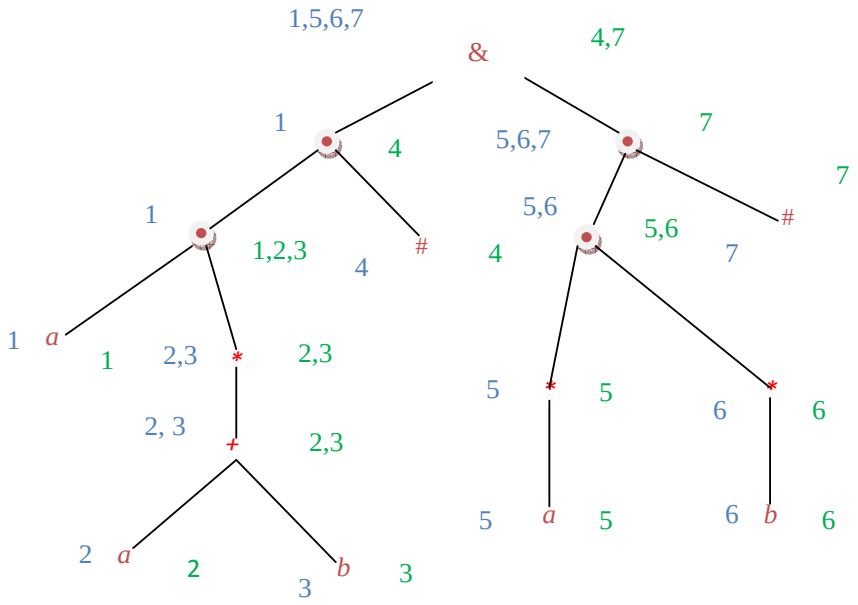


Figure 3.2.3: Syntax tree for the augmented *PRE* $r = (a(a|b)^*)\&(a^*b^*)$

$$followpos(1) \leftarrow \{2,3,4\}$$
$$followpos(2) \leftarrow \{2,3,4\}$$

$$followpos(3) \leftarrow \{2,3,4\}$$
$$followpos(5) \leftarrow \{5,6,7\}$$
$$followpos(6) \leftarrow \{6,7\}$$
$$followpos(4) \leftarrow followpos(7) \leftarrow \emptyset$$

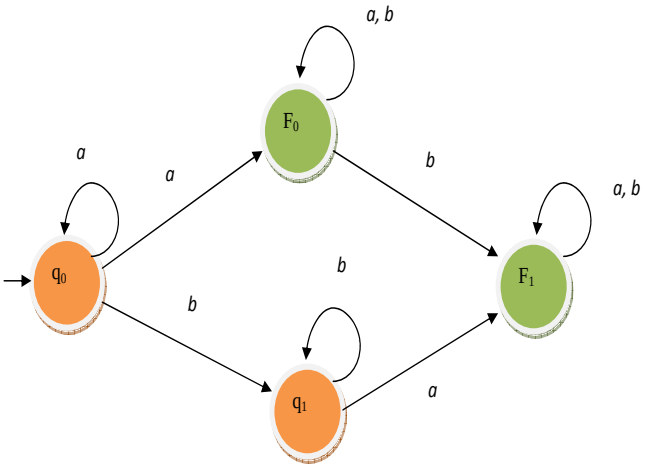


Figure 3.2.4: *NFA* for *PRE* $r = (a(a|b)^*)\&(a^*b^*)$

Figure 3.2.4 delineates the *NFA* $M=(\{q_0,q_1,F_0,F_1\},\{a,b\},\delta,q_0,\{F_0,F_1\})$ corresponding to *SRE r*.

3.3 Results And Discussion

Theorem 3.3.1 *If C is the number of occurrence of the concatenation operator in PRE r, then the NFA obtained using Estrade et al.'s [6] methodology requires* $2^{2|r|-3C}$ *states in the worst case.*

Proof. Estrade *et al.*'s methodology [6] converts *r* into *PFA* using the modified Thompson's construction. The number of states of the *PFA* using the modified Thompson's construction is equal to $2|r|-3C$. *PFA* with *n* states can be converted into *NFA* using the subset construction that requires 2^n states in the worst case [6]. Hence, *PFA* with $2|r|-3C$ states can be converted into *NFA* using $2^{2|r|-3C}$ states in the worst case. □

Theorem 3.3.2 *If m denotes the number of instances of symbols, and s denotes the number of shuffle operators in the SRE r, then ε-free NFA can be constructed using the proposed algorithm requiring 2^{m+s+1} states in the worst case.*

Proof. In the proposed algorithm, the syntax tree is constructed such that all symbols and #'s appear as the leaf nodes. Each state can be constructed using the positions assigned to the leaf nodes. Total number of leaf nodes in the syntax tree is $(m+s+1)$. Maximum 2^{m+s+1} state can be constructed using the positions of the syntax tree. Hence using the proposed algorithm, the worst case state complexity of the *NFA* is 2^{m+s+1}. □

Theorem 3.3.3 *The state complexity of the proposed algorithm is less than the state complexity of the Estrade et al.'s methodology [6] for the conversion of SRE to NFA.*

Proof. *SRE r* with m, u, k, s and c represent instances of symbols, union, Kleene closure, shuffle and concatenations respectively. The worst case state complexity of the proposed algorithm and Estrade *et al.*'s methodology are 2^{m+s+1} and $2^{2(m+u+k+s)-C}$ respectively. That the proposed algorithm will generate *NFA* with fewer states can be proved by the relation:

$$m+s+1 < 2(m+u+k+s)-C \Rightarrow 1 < m+2u+2k+s-C$$

Each concatenation operation is performed between two symbols. Similarly, n concatenation can be performed with at least $(n+1)$ symbols, which means $m-C \geq 1$.

Hence the relation $1 < m+2u+2k+s-C$ always holds, and the worst case state complexity of the proposed algorithm is always less than the Estrade *et al.*'s methodology. □

Table 3.3.1: Comparison between Estrade *et al.*'s approach and the proposed algorithm

Criterion	**Estrade *et al.* methodology** [6]	**Proposed Algorithm**
Worst case state complexity	$2^{2\|r\|-3C}$	2^{m+s+1}
Output	*ε-NFA*	ε-free *NFA*
Conversion Chain	$PRE \rightarrow PFA \rightarrow NFA \rightarrow DFA$	$SRE \rightarrow NFA \rightarrow DFA$

Worst-case state complexity of the ε-*NFA* is $2^{2|r|-3C}$ using the Estrade *et al.*'s methodology [6]. The pattern matching will decelerate if the *NFA* consists of ε-transitions. Using the proposed algorithm, ε-free *NFA* is generated. The worst case state complexity of the *NFA* is 2^{m+s+1}. Table 3.3.1 describes the differences in the proposed approach and Estrade *et al.*'s methodology [6].

3.4 CONCLUSION

In this chapter, an algorithm is designed for the metamorphosis of *SREs* to ε-free *NFAs* without using any intermediate steps. Using the proposed approach, a svelte *NFA* is generated. The number of states of the *NFA* generated using the proposed approach is smaller than the number of states of the *NFA* generated using the existing approaches in the literature. Further, if $\forall(r_i \& r_j) \in r \wedge ((symbols(r_i) \cap symbols(r_j)) \neq \emptyset)$, then it is implausible to generate a *DFA* directly from the *SRE*. In the future, this work can be extended for the metamorphosis of *PREs* to *NFAs*.

Chapter 4

A NOVEL ALGORITHM FOR THE CONVERSION OF PARALLEL REGULAR EXPRESSIONS TO NON-DETERMINISTIC FINITE AUTOMATA

4.1 INTRODUCTION

Estrade *et al.* [6] calibrate the metamorphosis of Parallel Regular Expressions (*PREs*) to Deterministic Finite Automata (*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 [5]. $\lambda-$transitions represent the joining of two languages using a shuffle operator. The *FLAT* tool kit [28] is designed for the metamorphosis of *PREs* to *DFAs* using the intermediate steps of Parallel Finite Automaton (*PFA*) and *NFA*.

In the direct conversion method [20], *REs* can be converted into *DFAs*. Given *RE* r over an alphabet Σ with length n. Let m depicts the number of instances of symbols from Σ in r. Table 4.1.1 narrates the output and number of states generated by Thompson's construction [43], Glushkov automata [72] and the direct conversion method [20]. 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 [20] for the metamorphosis of *PREs* to *NFAs*.

In this chapter, we concoct an algorithm for the metamorphosis of *PREs* to ε-free *NFAs*. For a given parallel regular expression r, let m be the number of symbols that

[1]Results of this chapter are published in *Applied Mathematics & Information Sciences*, Vol. 8(1), PP.: 95-105, 2014.

Table 4.1.1: Comparison between direct conversion, Thompson's construction and Glushkov automata

Criterion	**Thompson's Construction [43]**	**Glushkov Automata [72]**	**Direct Conversion [20]**
Output	ε-*NFA*	*NFA*	*DFA*
Number of states	$\|Q\| \leq 2n$	$\|Q\| = m+1$	$\|Q\| \leq m+1$

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 proposed algorithm. In the earlier existing approaches, the number of states of the non-deterministic finite automaton in the worst case scenario is equal to $2^{2|r|-3C}$.

4.2 PRELIMINARIES

DEFINITION 4.2.1. 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 4.2.2. *Firstpos(r)* [3] 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 4.2.3. *Lastpos(r)* [3] 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 4.2.4. *Nullable* [3] 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 4.2.5. *followpos(i)* [1] 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 4.2.6. *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 4.2.1. Given a particular node with the *PRE*, following are the various shuffle flag value depending upon *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 4.2.7. *shuffle-first(r)* of a *PRE r* consists of set of positions calculated as follows:

1. $shuffle - first(r) \leftarrow \{\emptyset\}$

2. If $((r_i\#_s\&r_j\#_s) \in r)$

$$shuffle - first(r) \leftarrow shuffle - first(r) \cup \mathbf{Min}(firstpos(r_i\#_s)) \cup \mathbf{Min}(firstpos(r_j\#_s))$$

DEFINITION 4.2.8.*Shuffle-last(r)* of a *PRE r* consists of set of positions calculated as follows:

1. $shuffle - last(r) \leftarrow \{\emptyset\}$

2. If $((r_i\#_s\&r_j\#_s) \in r)$

$$shuffle - last(r) \leftarrow shuffle - last(r) \cup \mathbf{Max}(lastpos(r_i\#_s)) \cup \mathbf{Max}(lastpos(r_j\#_s))$$

EXAMPLE 4.2.2. Given *PRE* $r = ((ab)\#_s\&(cd)\#_s)\#$

Augmented *PRE* $r' = ((ab)\#_s\&(cd)\#_s)\#$

$$shuffle - first(r) \leftarrow shuffle - first(r) \cup \mathbf{Min}(firstpos(ab\#_s)) \cup \mathbf{Min}(firstpos(cd\#_s))$$
$$\leftarrow \{1,4\}$$
$$shuffle - last(r) \leftarrow shuffle - last(r) \cup \mathbf{Max}(lastpos(ab\#_s)) \cup \mathbf{Max}(lastpos(cd\#_s))$$
$$\leftarrow \{3,6\}$$

followpos of $\#_s$ are taken as $\emptyset$. Now the question arises: which symbol is the next 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, as discussed below.

DEFINITION 4.2.9. *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 $((\varepsilon \in r_i) \wedge (\varepsilon \in r_j))$

   $$IF \leftarrow position_{\#_s}(r_i\#_s) \cup position_{\#_s}(r_j\#_s)$$

   $$immediate-follow(IF) \leftarrow immediate-follow(IF) \cup firstpos(r_i\#_s) \cup firstpos(r_k) \cup position_{\#_s}(r_j\#_s)$$

2. Let $((r_i\#_s \& r_j\#_s) \in r)$

   If $((\varepsilon \in r_j\#_s) \wedge (\varepsilon \in r_i))$

   $$IF \leftarrow position_{\#_s}(r_i\#_s) \cup position_{\#_s}(r_j\#_s)$$

   $$immediate-follow(IF) \leftarrow immediate-follow(IF) \cup firstpos(r_j\#_s) \cup position_{\#_s}(r_i\#_s) \cup firstpos(r_k)$$

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 4.2.3. 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 4.2.4. Given the *PRE* $r \leftarrow ((ab)^*\#_s\&(cd)\#_s)e\#$

Augmented *PRE* with their position is $((a_1b_2)^*\#_{s3}\&(c_4d_5)\#_{s6})e_7\#_8$

$immediate-follow(3,6) \leftarrow \{1,6,7\}$

EXAMPLE 4.2.5. Given the *PRE* $r \leftarrow ((ab)\#_s\&(cd)\#_s)^*e\#$

Augmented *PRE* with their position is $((a_1b_2)\#_{s3}\&(c_4d_5)\#_{s6})^*e_7\#_8$

$immediate-follow(3,6) \leftarrow \{1,4,7\}$

EXAMPLE 4.2.6. Given the *PRE* $r \leftarrow ((ab)^*\#_s\&(cd)^*\#_s)e\#$

Augmented *PRE* with their position is $((a_1b_2)^*\#_{s3}\&(c_4d_5)^*\#_{s6})e_7\#_8$

$immediate-follow(3,6) \leftarrow \{1,4,7\}$

DEFINITION 4.2.10. *Separator* is a function 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 4.2.11. *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$((shuffle(r_i) = true) \vee (shuffle(r_j) = true))$

   $Separator - first(r) \leftarrow Separator - first(r) \cup \mathbf{Min}(firstpos(r_i))$

   $\cup \mathbf{Min}(firstpos(r_j))$

3. Let $(r_i r_j) \in r$

   If$((shuffle(r_i) = true) \wedge (shuffle(r_j) = true) \wedge (nullable(r_j) = true))$

   $Separator - first(r) \leftarrow Separator - first(r) \cup \mathbf{Min}(firstpos(r_i))$

   $\cup \mathbf{Min}(firstpos(r_j))$

DEFINITION 4.2.12. *Separator-last(r)* of a *PRE r* is calculated as follows:

1. $Separator\text{-}last(r) \leftarrow \emptyset$

2. Let $(r_i|r_j) \in r$

   If$((shuffle(r_i) = true) \vee (shuffle(r_j) = true))$

   $$Separator-last(r) \leftarrow Separator-last(r) \cup \mathbf{Max}(lastpos(r_i)) \cup \mathbf{Max}(lastpos(r_j))$$

3. Let $(r_i r_j) \in r$

   If$((shuffle(r_i) = true) \wedge (shuffle(r_j) = true) \wedge (nullable(r_j) = true))$

   $$Separator-last(r) \leftarrow Separator-last(r) \cup \mathbf{Max}(lastpos(r_i)) \cup \mathbf{Max}(lastpos(r_j))$$

EXAMPLE 4.2.7. 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 4.2.8. Given the augmented *PRE* $r = ((a_1\#_{s2})\&(b_3\#_{s4})).((c_5\#_{s6})\&(d_7\#_{s8}))$ at a node n.

$$separator(n) \leftarrow true$$
$$separator-first(r) \leftarrow \{1,5\}$$
$$separator-last(r) \leftarrow \{4,8\}$$

4.3 PROPOSED ALGORITHM

In the direct conversion [20] 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, an algorithm (*MPRENFA*) is proposed for the metamorphosis of *PREs* to *NFAs*. 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 4.2.2 to 4.2.5 with meagre modification. These variations occur due to the shuffle operators. *Shuffle*, *shuffle-first*, *shuffle-last*, *separator*, *separator-first*, *separator-last* and *immediate-follow* are determined using the definitions 4.2.6 to 4.2.12.

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 4.3.1. 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 Figure 4.3.1 is constructed for the augmented *PRE* r'. During traversal of the syntax tree in post-order, different values are computed

using the definitions 4.2.2 to 4.2.12.

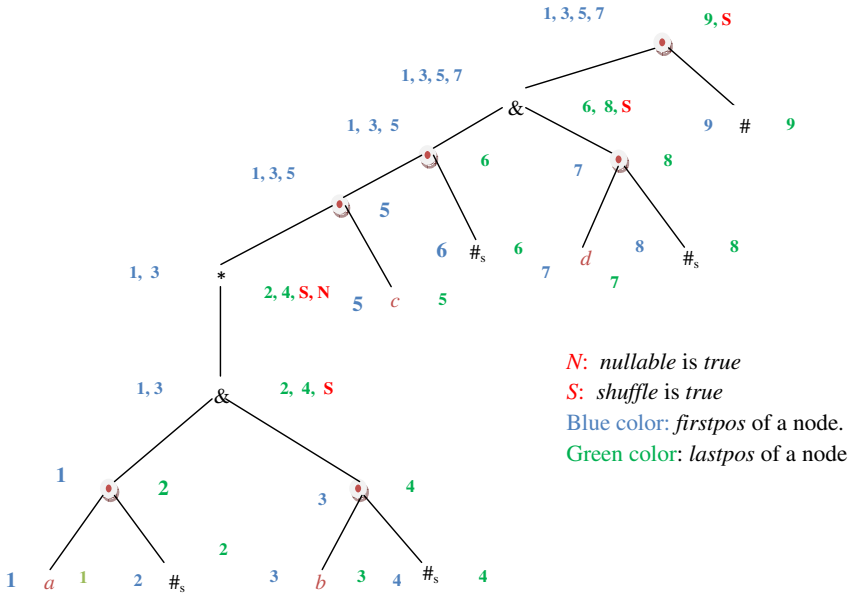


Figure 4.3.1: Syntax tree for the augmented *PRE* $r = ((a\#_s \& b\#_s)^* c\#_s)\& d\#_s\#$

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 the procedure *Follow_Calc*. In 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 values from a set. The complete procedure is explained with the help of example 4.3.2. *symbol(i)* depicts the symbol at i^{th} position. *IF* depicts the positions of a pair of $\#_s$.

EXAMPLE 4.3.2. On applying *Follow_Calc*(r') to the Example 4.3.1

Using algorithm 4.3.2, following values are determined during traversal of the syntax tree:

$$immediate-follow(2,4) \leftarrow \{1,3,5\}$$

$$immediate\text{-}follow(6,8) \leftarrow \{9\}$$

$$shuffle-first \leftarrow \{1,3\}$$

$$shuffle-last \leftarrow \{2,4\}$$

$$followpos(1) \leftarrow \{2\}\ , followpos(3) \leftarrow \{4\}$$

$$followpos(5) \leftarrow \{6\}\ , followpos(7) \leftarrow \{8\}$$

Using different values determined from algorithm *Follow_Calc*, we can convert *PRE* into $\varepsilon-$free *NFA*. The proposed algorithm for the construction of *NFA* is given in algorithm 4.3.5. 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 symbols 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 4.4 by a numerical example.

Algorithm 4.3.1: *MPRENFA*(*r, NFA*)

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

```
1 for ∃((r_i&r_j) ∈ r) do
2 |                                              ▷ Augmented PRE
3 |   r_i ← r_i#_s
4 |   r_j ← r_j#_s
5 end
6 r' ← r#
7 Assign position to the symbols of the augmented PRE
8 Follow_Calc(r')                          ▷ Call to procedure Follow_Calc
9 Create-NFA()                             ▷ Call to procedure Create-NFA
```

Algorithm 4.3.2: *Procedure Follow_Calc(r')*

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

Construct a syntax tree for r'
Repeat step *3* to *60* for each node n during the post-order traversal of the syntax tree
if $(((n = a) \wedge (a \in \Sigma)) \vee ((n = \#) \vee (n = \#_s)))$ **then**
    $firstpos(n) \leftarrow lastpos(n) \leftarrow position(n)$ ▷ Leaf Node
    $nullable(n) \leftarrow shuffle(n) \leftarrow false$
    $separator(n) \leftarrow false$
end
if $(n = .)$ **then**
    Concatenation-traverse(); ▷ Concatenation operator, call to Algorithm 4.3.3
end
if $(n = *)$ **then**
    $firstpos(n) \leftarrow firstpos(L)$ ▷ Kleene closure
    $lastpos(n) \leftarrow lastpos(L)$
    $nullable(n) \leftarrow true$
    $separator(n) \leftarrow separator(L)$
    $shuffle(n) \leftarrow shuffle(L)$
    $IF \leftarrow \emptyset$
    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 element 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
                    $immediate - follow(IF) \leftarrow immediate - follow(IF) \cup firstpos(L)$
                end
            end
        else
            $immediate - follow(IF) \leftarrow immediate - follow(IF) \cup firstpos(L)$
        end
    end
end

Algorithm 4.3.2: *Procedure Follow_Calc(r')* **Continued Algorithm 4.3.2**

```
42 if (n = |) then
43 |   firstpos(n) ← firstpos(L) ∪ firstpos(R)                ▷ Union Node
44 |   lastpos(n) ← lastpos(L) ∪ lastpos(R)
45 |   if ((nullable(L) = false) ∧ (nullable(R) = false)) then
46 |   |   nullable(n) ← false
47 |   else
48 |   |   nullable(n) ← true
49 |   end
50 |   if ((shuffle(L) = true) ∨ (shuffle(R) = true)) then
51 |   |   shuffle(n) ← separator(n) ← true
52 |   |   separator − first ←
   |   |   separator − first ∪ Min(firstpos(L)) ∪ Min(firstpos(R))
53 |   |   separator − last ←
   |   |   separator − last ∪ Max(firstpos(L)) ∪ Max(firstpos(R))
54 |   else
55 |   |   shuffle(n) ← false
56 |   |   separator(n) ← false
57 |   end
58 end
59 if (n = &) then
60 |   Shuffle-traverse();          ▷ Shuffle operator, call to Algorithm 4.3.4
61 end
```

Algorithm 4.3.3: *Procedure Concatenation_traverse*

Input: *An Augmented PRE r'*
Output: *Followpos*,*shuffle-first*,*shuffle-last*, *immediate-follow*,*separator-first*, *separator-last* at Concatenation operator

```
1  firstpos(n) ← firstpos(L)
2  lastpos(n) ← lastpos(R)
3  if (nullable(L)=true) then
4  |   firstpos(n) ← firstpos(n) ∪ firstpos(R)
5  end
6  if (nullable(R)=true) then
7  |   lastpos(n) ← lastpos(n) ∪ lastpos(L)
8  end
9  if (nullable(L) = true) ∧ ((nullable(R) = true) ∨ (R = #_s)) then
10 |   nullable(n) ← true
11 else
12 |   nullable(n) ← false
13 end
```

Algorithm 4.3.3: *Procedure Concatenation_traverse* Continued Algorithm 4.3.3

```
14 if (shuffle(R) = true) then
15 |   shuffle(n) ← true
16 else
17 |   if ((shuffle(L) = true) ∧ ((R = #_s) ∨ (nullable(R) = true))) then
18 |   |   shuffle(n) ← true
19 |   else
20 |   |   shuffle(n) ← false
21 |   end
22 end
23 if ((shuffle(L) = true) ∧ (shuffle(R) = true)) then
24 |   if (nullable(R) = true) then
25 |   |   separator(n) ← true
26 |   |   separator−first ← separator−first ∪ Min(firstpos(L))
   |   |                                 ∪ Min(firstpos(R))
27 |   |   separator−last ← separator−last ∪ Max(lastpos(L))
   |   |                                 ∪ Max(lastpos(R))
28 |   end
29 else
30 |   separator(n) ← false
31 end
32                                                  ▷ IF consists of positions of #_s
33 IF ← ∅
34 for (i ∈ lastpos(L)) do
35 |   if (symbol(i) ≠ #_s) then
36 |   |   followpos(i) ← followpos(i) ∪ firstpos(R)
37 |   else
38 |   |   IF ← IF ∪ i
39 |   end
40 end
41 if (IF ≠ ∅) then
42 |                                  ▷ Assume sf elements in separator-first
43 |   if (separator(n) = true) then
44 |   |   for (i = 1; i = sf; i = i + 1) do
45 |   |   |   for (z ∈ IF) do
46 |   |   |   |   T ← ∅
47 |   |   |   |   if (separator−first[i] ≤ z ≤ separator−last[i]) then
48 |   |   |   |   |   T ← T ∪ z
49 |   |   |   |   end
50 |   |   |   end
51 |   |   |   immediate−follow(T) ← immediate−follow(T) ∪ firstpos(R)
52 |   |   end
53 |   else
54 |   |   immediate-follow(IF) ← immediate-follow(IF) ∪ firstpos(R)
55 |   end
56 end
```

Algorithm 4.3.4: Procedure *Shuffle_traverse*

Input: *An Augmented PRE r′*
Output: *Followpos*,*shuffle-first*,*shuffle-last*, *immediate-follow*,*separator-first*, *separator-last* at *shuffle* operator

```
1  firstpos(n) ← firstpos(L) ∪ firstpos(R)
2  lastpos(n) ← lastpos(L) ∪ lastpos(R)
3  shuffle(n) ← true
4  if (nullable(L) = true) ∧ (nullable(R) = true) then
5  |  nullable(n) ← true
6  else
7  |  nullable(n) ← false
8  end
9  flag ← false
10 if (shuffle(L) = false) then
11 |                                   ▷ For avoiding duplicate entry in shuffle-first
12 |  for (z = 1; z < m; z = z + 1) do
13 |  |                                ▷ Assume m shuffle-first entry
14 |  |  if (Min(firstpos(L)) ≤ shuffle − first[z] ≤ Max(lastpos(L))) then
15 |  |  |  flag ← true
16 |  |  |  break
17 |  |  end
18 |  end
19 |  if (flag ≠ true) then
20 |  |  shuffle − first ← shuffle − first ∪ Min(firstpos(L))
21 |  |  shuffle − last ← shuffle − last ∪ Max(lastpos(L))
22 |  end
23 end
24 flag ← false                         ▷ For avoiding duplicate entry in shuffle-first
25 if (shuffle(R) = false) then
26 |  for (z = 1; z < m; z = z + 1) do
27 |  |  if ((Min(firstpos(R)) ≤ shuffle − first[z] ≤ (Max(lastpos(R))) then
28 |  |  |  flag ← true
29 |  |  |  break
30 |  |  end
31 |  end
32 |  if (flag ≠ true) then
33 |  |  shuffle − first ← shuffle − first ∪ Min(firstpos(R))
34 |  |  shuffle − last ← shuffle − last ∪ Max(lastpos(R))
35 |  end
36 end
37 if ((separator(L) = true) ∨ (separator(R) = true)) then
38 |  separator(n) ← true
39 end
40 IF ← ∅                                ▷ Determine immediate-follow
```

Algorithm 4.3.4: Procedure *Shuffle_traverse* Continued Algorithm 4.3.4

```
41 if (i ∈ lastpos(n)) then
42     if (symbol(i) = #_s) then
43         IF ← IF ∪ i
44     end
45 end
46 if (IF ≠ ∅) then
47     immediate − follow(IF) ← ∅
48     if (nullable(L) = true) then
49         immediate − follow(IF) ← immediate − follow(IF) ∪ firstpos(L)
50         if (nullable(R) = false) then
51             immediate − follow(IF) ← immediate − follow(IF) ∪ firstpos(R)
52         end
53     end
54     if (nullable(R) = true) then
55         immediate − follow(IF) ← immediate − follow(IF) ∪ firstpos(R)
56         if (nullable(L) = false) then
57             immediate − follow(IF) ← immediate − follow(IF) ∪ firstpos(L)
58         end
59     end
60 end
```

4.4 NUMERICAL EXAMPLE

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

EXAMPLE 4.4.1. Given the *PRE* $r = (a\&b)^*(c\&d)^*$

$$followpos(1) \leftarrow \{2\},\ followpos(3) \leftarrow \{4\}$$

$$followpos(5) \leftarrow \{6\},\ followpos(7) \leftarrow \{8\}$$

$$shuffle - first(r) \leftarrow \{1,3,5,7\}$$

$$shuffle - last(r) \leftarrow \{2,4,6,8\}$$

Algorithm 4.3.5: *Procedure Create-NFA*

Input: *Augmented PRE and different values calculated using Follow_Calc(r)*
Output: An equivalent *NFA* corresponding to *PRE*

1 **Starting state**: $q_0 \leftarrow firstpos(root)$
2 $Q \leftarrow \{q_0\}$ and make q_0 as unmarked state. **Continued...**

Algorithm 4.3.5: *Procedure Create-NFA* **Continued...**

3 **for** ($\exists$ *unmarked* $q \in Q$) **do**
4 Choose an unmarked state q and mark it.
5 **for** ($\exists a \in \Sigma$) **do**
6 ▷ For each value of *shuffle-first* and *shuffle-last*
7 **for** ($i = 1; i \leq n; i = i + 1$) **do**
8 $epos \leftarrow lpos \leftarrow qtrans \leftarrow \emptyset$
9 **for** ($\exists j \in q$) **do**
10 **if** ($shuffle - first[i] \leq j \leq shuffle - last[i]$) **then**
11 **if** ($symbol(j) = a$) **then**
12 $qtrans \leftarrow qtrans \cup followpos(j)$
13 $lpos \leftarrow j$
14 **else**
15 **if** ($symbol(j) \neq \#$) **then**
16 $epos \leftarrow epos \cup j$
17 **end**
18 **end**
19 **end**
20 **end**
21 **if** ($qtrans = \emptyset$) **then**
22 go to step 7
23 **end**
24 **for** ($j = 1; j \leq m; j = j + 1$) **do**
25 **if** ($separator - first[j] \leq lpos \leq separator - last[j]$) **then**
26 **break**
27 **end**
28 **end**
29 **for** ($\exists pos \in epos$) **do**
30 **if** ($separator - first[j] > pos$) $\vee$ ($separator - last[j] < pos$) **then**
31 $epos = epos - pos$
32 **end**
33 **end**
34 $qtrans \leftarrow qtrans \cup epos$
35 $IF \leftarrow \emptyset$
36 **for** ($\exists pos \in qtrans$) **do**
37 **if** ($symbol(pos) = \#_s$) **then**
38 $IF \leftarrow IF \cup pos$
39 **end**
40 **end**
41 **if** ($immediate - follow(IF) \neq \emptyset$) **then**
42 $qtrans \leftarrow qtrans - IF \cup immediate - follow(IF)$
43 $trans[q, a] = qtrans$
44 $Q \leftarrow Q \cup qtrans$
45 **end**
46 **end**
47 **end**
48 **end**
49 **Final state:** $F = \{\forall q_f | (lastpos(root) \subseteq q_f) \wedge (q_f \in Nstates)\}$

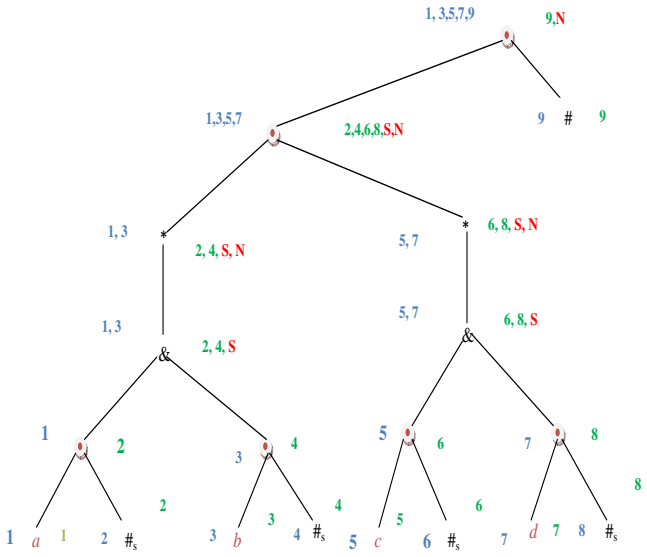


Figure 4.4.1: Syntax tree for the augmented *PRE* $r = (a\#_s\&b\#_s)^*(c\#_s\&d\#_s)^*$ using the proposed approach

Figure 4.4.1 delineates the syntax tree corresponding to r.

$immediate-follow(2,4) \leftarrow \{1,3,5,7,9\}$

$immediate-follow(6,8) \leftarrow \{5,7,9\}$

$separator-first \leftarrow \{1,5\}$

$separator-last \leftarrow \{4,8\}$

***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 $separator-first \leftarrow \{1,5\},\ separator-last \leftarrow \{4,8\},\ and$

$epos \leftarrow \{3\}$

$qtrans \leftarrow followpos(1) \cup epos \leftarrow \{2,3\}$

$trans(q_0,a) \leftarrow \{2,3\} \leftarrow q_1$

$i \leftarrow 2,\ i \leftarrow 3,\ i \leftarrow 4$

$qtrans \leftarrow \emptyset$

Symbol b

$i \leftarrow 1$

$qtrans \leftarrow \emptyset$

$i \leftarrow 2$

$qtrans \leftarrow followpos(3) \leftarrow \{4\},\ epos \leftarrow \{1,5,7\}$

Using $separator-first \leftarrow \{1,5\},\ separator-last \leftarrow \{4,8\},\ and$

$epos \leftarrow \{1\}$

$qtrans \leftarrow followpos(3) \cup epos \leftarrow \{1,4\}$

$trans(q_0,b) \leftarrow \{1,4\} \leftarrow q_2$

$i \leftarrow 3, i \leftarrow 4$

$qtrans \leftarrow \emptyset$

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 $separator-first \leftarrow \{1,5\},\ separator-last \leftarrow \{4,8\},\ and$

$epos \leftarrow \{7\}$

$qtrans \leftarrow followpos(5) \cup epos \leftarrow \{6,7\}$

$trans(q_0,c) \leftarrow \{6,7\} \leftarrow q_3$

$i \leftarrow 4$

$qtrans \leftarrow \emptyset$

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 $separator-first \leftarrow \{1,5\},\ separator-last \leftarrow \{4,8\},\ and$ $epos \leftarrow \{5\}$

$qtrans \leftarrow followpos(7) \cup epos \leftarrow \{5,8\}$

$trans(q_0,d) \leftarrow \{5,8\} \leftarrow q_4$

Reading of symbols on q_1

Symbol a

$i \leftarrow 1$

$qtrans \leftarrow followpos(1) \leftarrow \{2\},\ epos \leftarrow \{4\}$

Using $separator-first \leftarrow \{1,5\},\ separator-last \leftarrow \{4,8\},\ and$ $epos \leftarrow \{4\}$

$qtrans \leftarrow followpos(1) \cup epos \leftarrow \{2,4\}$

$trans(q_2,a) \leftarrow immediate-follow\{2,4\} \leftarrow \{1,3,5,7,9\} \leftarrow q_0$

$i \leftarrow 2,\ i \leftarrow 3,\ i \leftarrow 4$

$qtrans \leftarrow \emptyset$

Symbol b

$i \leftarrow 1$

$qtrans \leftarrow \emptyset$

$i \leftarrow 2$

$qtrans \leftarrow followpos(3) \leftarrow \{4\},\ epos \leftarrow \{2\}$

Using $separator-first \leftarrow \{1,5\},\ separator-last \leftarrow \{4,8\},\ and$ $epos \leftarrow \{2\}$

$qtrans \leftarrow followpos(3) \cup epos \leftarrow \{2,4\}$

$trans(q_1,b) \leftarrow immediate-follow\{2,4\} \leftarrow \{1,3,5,7,9\} \leftarrow q_0$

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 Figure 4.4.2.

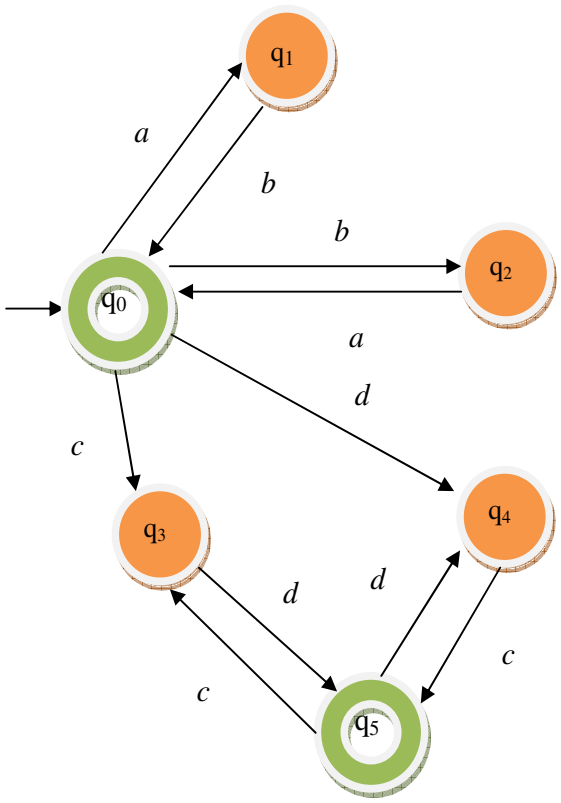


Figure 4.4.2: *DFA* for *PRE* $r = (a\&b)^*(c\&d)^*$

4.5 Results and Discussion

Using Estrade *et al.*'s methodology [6], *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. Figure 4.5.1 delineates the *PFA* corresponding to *PRE* $a\&b^*$ using Estrade *et al.*'s methodology. Figure 4.5.2 delineates the *DFA* corresponding to the *PRE* $a\&b^*$ using the proposed algorithm. Table 4.5.1 depicts the differences between the proposed algorithm and Estrade *et al.*'s methodology.

Table 4.5.1: Comparison between Estrade *et al.*'s methodology and proposed algorithm

Criterion	**Estrade *et al.* [6]**	**Proposed Algorithm**
Output	*ε-NFA*	*ε-free NFA*
Conversion chain	$PRE \rightarrow PFA \rightarrow \varepsilon - NFA$	$PRE \rightarrow NFA$
Number of states	$2^{2\|r\|-3C}$ states	2^{m+1}

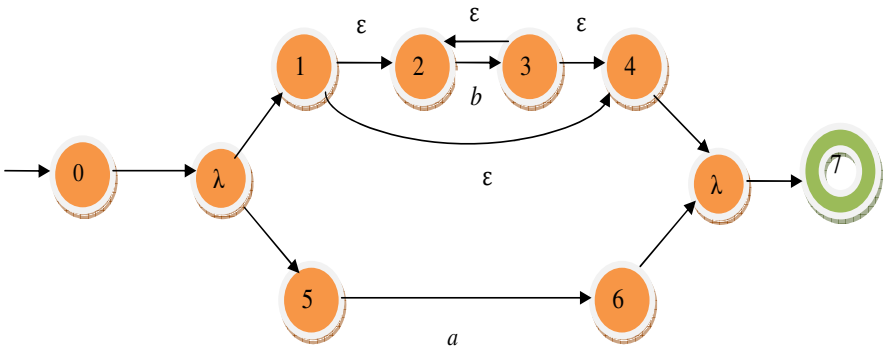


Figure 4.5.1: *PFA* for $r = a\&b^*$ using Estrade *et al.*'s methodology

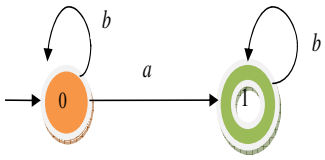


Figure 4.5.2: DFA for $r = a\&b^*$ using the proposed approach

THEOREM 4.5.1. 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. □

4.6. CONCLUSION

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 [6]. The major benefit of the proposed algorithm is the production of a svelte *NFA*. Another major benefit of the proposed 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.

Chapter 5

STATE COMPLEXITY OF THE NON-DETERMINISTIC FINITE AUTOMATA OBTAINED FROM PARALLEL REGULAR EXPRESSIONS USING PARTIAL DERIVATIVES

5.1 INTRODUCTION

Thompson [43] proposed the first technique for the conversion of Regular Expression (*RE*) to Non-deterministic Finite Automaton (*NFA*). Sippu and Soisalon-Soininen [65] proposed an approach, using which a smaller *NFA* can be generated than the Thompson's construction. The *NFA* generated using both the above approaches involves ε-transitions. If the *NFA* contains ε-transitions, the pattern matching will be delayed due to the ε-transitions involved in the *NFA*. ε-free *NFA* can be constructed using different techniques named as Glushkov automata, follow automata and partial derivative automata [44, 68, 72]. Partial derivative automata generate a smaller ε-free *NFA* having at most $(m+1)$ states and m^2 transitions. Champarnaud and Ziadi [39] proposed an improved algorithm with $O(n^2)$ time complexity for the construction of partial derivative automata.

Researchers have proposed different methodologies for the conversion of Parallel Regular Expressions (*PREs*) to *NFAs* having the number of states $O(2^r)$. Estrade *et al.* [6] investigated the conversion of *PREs* to Deterministic Finite Automata (*DFAs*). The proposed work of this chapter is to generalize the partial derivative automata for the conversion of *PRE* to ε-free *NFA*. This innovative approach is the generalization of the Antimirov partial derivatives for the conversion of *REs* to ε-free *NFAs*. Using this approach, the number of states of the *NFA* is reduced from exponential $O(2^r)$ to polynomial $O(m^{k+1})$.

[1]Results of this chapter are peer reviewed and accepted for publication in Chiang Mai Journal of Science

5.2 PRELIMINARY CONCEPT

DEFINITION 5.2.1. *First*(r) can be defined as a set consisting of the positions of the first symbols generated from *RE r*.

DEFINITION 5.2.2. Integer partition [27] of a positive integer n is a non-increasing sequence of positive integers $p_1, p_2, ..., p_n$ such that $n = p_1 + p_2 + ... + p_n$. Each p_i is called a part of the partition.

EXAMPLE 5.2.1. For $n = 5$, following seven partitions exist:

$i)5 \;\; ii)4+1 \;\; iii)3+2 \;\; iv)3+1+1 \;\; v)2+2+1 \;\; vi)2+1+1+1 \;\; vii)1+1+1+1+1$

DEFINITION 5.2.3. Sub-regular expression denotes a part of the *PRE* which does not contain any shuffle operator and it contain only the union, concatenation and Kleene closure operator.

DEFINITION 5.2.4. *PRE r* is in unit-level of shuffling if shuffled sub-regular expressions of r will not get shuffled with other sub-regular expressions.

DEFINITION 5.2.5. *PRE r* is in integration or multiple level of shuffling if shuffled sub-regular expressions of r will get shuffled with other sub-regular expressions.

EXAMPLE 5.2.2. $(a\&b)c|(a\&c)$ is an example of unit-level of shuffling. $(a\&b)\&c$ is an example of multiple level of shuffling.

Throughout this chapter, m and k denotes the total number of occurrence of symbols and level of shuffling in r. We use ordered integer partitions with a maximum value $(n-1)$ and order of integers play a significant role.

5.3 PARTIAL DERIVATIVE AUTOMATA

Derivatives of *RE* were introduced by Brzozowski [33] in 1964, applying this algorithm *RE* can be converted into *DFA*. Later on, Antimirov [68] proposed the concept of partial derivatives for the conversion of *RE* to *NFA*.

Consider a language L over an alphabet $\{a,b\}$ denoted by *RE r*. Then, according to

Antimirov [68] starting and final state of the partial derivative automata is determined using the following rules:

Starting state (q_0) and final state (F) of the *NFA* are defined as follow:

$$q_0 = r \text{ and } F = \{q \in Q | \varepsilon \in L(q)\}$$

EXAMPLE 5.3.1. Consider the *RE* $r = (a|b)^*aa(a|b)^*$ over an alphabet $\Sigma = \{a,b\}$. Using partial derivatives, *NFA* (illustrated in Figure 5.3.1) is computed as follow:

$$r = (a|b)^*aa(a|b)^* = 0$$

$$\partial_a((a|b)^*aa(a|b)^*) = (\partial_a(a|b)^*)(aa(a|b)^*) \cup \partial_a(aa(a|b)^*)$$
$$= (a|b)^*aa(a|b)^* \cup (a(a|b)^*) = 0 \cup 1$$

$$\partial_b((a|b)^*aa(a|b)^*) = (\partial_b(a|b)^*)(aa(a|b)^*) \cup \partial_b(aa(a|b)^*)$$
$$= (a|b)^*aa(a|b)^* \cup \emptyset = 0$$

$$\partial_a(a(a|b)^*) = (\partial_a(a)(a|b)^*) = (a|b)^* = 2$$

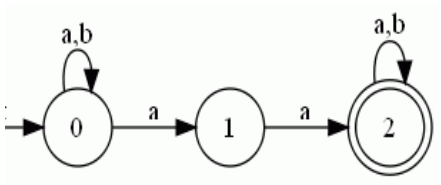


Figure 5.3.1: *NFA* for $r = (a|b)^*aa(a|b)^*$

$$\partial_b(a(a|b)^*) = \emptyset$$

$$\partial_a((a|b)^*) = (a|b)^* = 2$$

$$\partial_b((a|b)^*) = (a|b)^* = 2$$

PROPOSITION 5.3.1. Worst case *NFA* (i.e. *NFA* having $m+1$ states) occurs using partial derivatives, if m instances of symbols are connected by $(m-1)$ concatenation operators and no other operators are used.

Proof. Consider *RE* $r = a_1a_2...a_m$. Using partial derivative automata, $q_0 = r$.

$$\delta(q_0, a_1) = \partial_{a1}(r) = a_2a_3...a_m = q_1$$

$$\delta(q_1, a_2) = \partial_{a2}(q_1) = a_3a_4...a_m = q_2$$

$$\delta(q_{m\text{-}1}, a_m) = \partial_{am}(q_{m-1}) = \varepsilon = q_m$$

If m instance of symbols are connected by $(m-1)$ concatenation operators and no other operators are used, then using partial derivatives $(m+1)$ states are required. Next, we prove that the worst case cannot occur if r contains union or Kleene closure operator.

***RE r* contains Kleene closure:** Consider *RE* r is of the form $a_1a_2...a_{i-1}(a_i...a_j)^*a_{j+1}...a_m$. Using partial derivatives,

$$\delta(q_{i-1}, a_i) = \partial_{a_i}((a_i...a_j)^*a_{j+1}...a_m) = (a_{i+1}...a_j)(a_ia_{i+1}...a_j)^*a_{j+1}...a_m = q_i$$

$$\begin{aligned}\delta(q_i, a_{i+1}) &= \partial_{a_{i+1}}((a_{i+1}...a_j)(a_i...a_j)^*a_{j+1}...a_m) \\ &= (a_{i+2}...a_j)(a_ia_{i+1}...a_j)^*a_{j+1}...a_m = q_{i+1}\end{aligned}$$

$$\begin{aligned}\delta(q_{j-1}, a_j) &= \partial_{a_j}(a_j)(a_i...a_j)^*(a_{j+1}...a_m) \\ &= (a_ia_{i+1}...a_j)^*a_{j+1}...a_m = q_{i-1}\end{aligned}$$

Thus, each Kleene closure causes the number of states to be reduced by one. Hence, if r contains a Kleene closure, then it cannot have more than m states.

***RE r* contains union operator:** Consider *RE* r is of the form $a_1a_2...a_{i-1}(a_i...a_j|a_{j+1}...a_t)a_{t+1}...a_m$.

Using partial derivatives

$$\delta(q_{i-1}, a_i) = \partial_{a_i}((a_i...a_j|a_{j+1}...a_t)a_{t+1}...a_m) = (a_{i+1}...a_j)a_{t+1}...a_m = q_i$$

$$\delta(q_i, a_{i+1}) = \partial_{a_{i+1}}((a_{i+1}...a_j)a_{t+1}...a_m) = (a_{i+2}...a_j)a_{t+1}...a_m = q_{i+1}$$

$$\delta(q_{j-1}, a_j) = \partial_{a_j}((a_j)a_{t+1}...a_m) = a_{t+1}...a_m = q_j$$

Similarly,

$$\delta(q_{i-1}, a_{j+1}) = \partial_{a_{j+1}}((a_i...a_j|a_{j+1}...a_t)a_{t+1}...a_m) = (a_{j+2}...a_t)a_{t+1}...a_m = q_{j+1}$$

$$\delta(q_{t-1}, a_t) = \partial_{a_t}((a_t)a_{t+1}...a_m) = (a_{t+1}...a_m) = q_j$$

On reading a_j and a_t respectively from q_{j-1} and q_{t-1}, a new state q_j is generated. Hence, the number of states will be decreased by one. Each union operation causes the number of states to be decrease by one. Therefore, if *RE* contains union operator, it can have maximum m states using partial

derivatives. □

PROPOSITION 5.3.2. *NFA* obtained using Thompson's construction will have the minimum number of states, if *r* contains only concatenation operator.

Proof. Using Thompson's construction [43], each symbol, Kleene closure and union operator can be represented by the addition of two new states, but each concatenation operator is carried out by merging of two states. If *r* only contains $(m-1)$ concatenation operators, then the *NFA* constructed using Thompson's construction require $(m+1)$ states. Best case of the *NFA* (*NFA* having the minimum number of states) using Thompson's construction occurs if *r* contains only concatenation operators. □

5.4 State Complexity of NFA Obtained From PRE

Following are the properties of shuffle operator [5, 6]:

1. Identity law: $\varepsilon \,\&\, r = r$
2. Associative law: $(r_1 \,\&\, r_2) \,\&\, r_3 = r_1 \,\&\, (r_2 \,\&\, r_3)$
3. Commutative law: $(r_1 \,\&\, r_2) = (r_2 \,\&\, r_1)$
4. $\varepsilon \,\&\, \varepsilon = \varepsilon$
5. $(a \,\&\, b)^* = (a \mid b)^*$
6. $(r_1 \,\&\, r_2) \,\&\, (r_3 \,\&\, r_4) = (r_1 \,\&\, r_2 \,\&\, r_3 \,\&\, r_4)$

LEMMA 5.4.1. *For each symbol* $a \in \Sigma$ *and for regular terms x,y and z over* Σ*, following equations will hold:*

1. $\partial_a (x \,\&\, y) = (\partial_a (x) \,\&\, y) \cup (x \,\&\, \partial_a (y))$
2. $\partial_a ((x \,\&\, y) \,\&\, z) = (\partial_a (x) \,\&\, y \,\&\, z) \cup (x \,\&\partial_a (y) \,\&\, z) \cup (x \,\&\, y \,\&\, \partial_a (z))$

Proof.

1. Let $r = x \,\&\, y$ be a state at any moment of time. For each symbol, $a \in \Sigma$, we may or may not have input transitions from the state represented by $x \,\&\, y$.

   If $a \in first(x)$, then $\delta(n,a) = (\partial_a(x)\&y)$.

   Similarly, if $a \in first(y)$, then $\delta(n,a) = (\partial_a(y)\&x)$.

   Combining these using shuffle properties, we get

   $\partial_a\,(x \,\&\, y) = (\partial_a\,(x) \,\&\, y) \cup (x \,\&\, \partial_a\,(y))$.

2. $\partial_a\,((x \,\&\, y) \,\&\, z) = ((\partial_a\,(x \,\&\, y)) \,\&\, z) \cup ((\, x \,\&\, y) \,\&\, \partial_a\,(z))$

   $= (\partial_a\,(x) \,\&\, y \,\&\, z) \cup (\, x \,\&\, \partial_a(y) \,\&\, z) \cup ((\, x \,\&\, y) \,\&\, \partial_a\,(z))$

   $= (\partial_a\,(x) \,\&\, y \,\&\, z) \cup (\, x \,\&\partial_a\,(y) \,\&\, z) \cup (\, x \,\&\, y \,\&\, \partial_a\,(z))$ □

LEMMA 5.4.2. *Consider a state $n_1 = (x\&y)z$ such that $\varepsilon \notin L(x\&y)$. On reading symbol $a \in \Sigma$, we have*

1. If $\partial_a\,(x) = \emptyset$ and $\partial_a\,(y) \neq \emptyset$, then $\delta(n_1,a) = (x\&\partial_a(y))z = n_2$

2. If $\partial_a\,(x) = \partial_a\,(y) = \emptyset$, then $\delta(n_1,a) = \emptyset$

Proof.

1. $a \notin first(x)$ implies $\partial_a\,(x) = \emptyset$ and $a \in first(y)$ implies $\partial_a\,(y) \neq \emptyset$.

   Given $\varepsilon \notin L(x \,\&\, y)$ and $\partial_a\,(x) = \emptyset$, implies that symbol a, cannot be generated from x and z. Using partial derivatives $\delta(n_1,a) = (x\&\partial_a(y))z = n_2$

2. If $\partial_a\,(x) = \partial_a\,(y) = \emptyset$ means $a \notin first(x)$ and $a \notin first(y)$. It means the symbol a, cannot be the first symbol generated from the *PRE* $x\&y$. Using partial derivatives $\delta(n_1,a) = \emptyset$ □

Table 5.4.1 and Table 5.4.2 show the precedence and associativity of operators.

Table 5.4.1: Operator's precedence and associativity in *PRE*

S.No.	Operator Name	Associativity	Example	Preference
1	*Shuffle*	*Right to Left*	$a\&(b\|c)$	*shuffle*
2	*Union*	*Right to Left*	$a^* \mid bc$	*Union*
3	*Kleene Closure*	*Right to Left*	a^*b	*Kleene closure*
4	*Concatenation*	*Left to Right*	abc	*First Concatenation*

Table 5.4.2: Operator precedence while one operator distributed over other operators

S.No.	Description	Expression Type	Example(s)	Preference
1	*Shuffle operator distributed over other operators*	$r_1\&(r_2 \mid r_3)$ $(r_1)^*\&(r_2)^*$ $(r_1r_2)\&(r_3r_4)$	$a\&(b\|c)$ $(a\&b)^*\&c^*$ $(ab)\&(cd)$	*shuffle*
2	*Union operator distributed over other operators*	$r_1 \mid (r_2\&r_3)$ $r_1 \mid r_2$ $(r_1^*) \mid (r_2^*)$	$c \mid (a \& b)$ $(ab)\mid (cd)$ $a^* \mid c^*$	*Union*
3	*Kleene closure operator over other operators*	$(r_1\&r_2)^*$ $(r_1r_2)^*$ $(r_1 \mid r_2)^*$	$(a\&b)^*$ $(ab)^*$ $(ab\|bc)^*$	*Kleene closure*
4	*Concatenation*	$(r_1 \mid r_2)c$ $(r_1\&r_2)c$ $(r_1 \mid r_2)^*r_3$	$(a \mid b)c$ $(a\&b)c$ $(a\&b)^*c$	*Union* *Shuffle* *Kleene closure*

LEMMA 5.4.3. *Consider a state $n_1 = (x\&y)z$ such that $\varepsilon \in L(x\&y)$. On reading of symbol $a \in \Sigma$, we have*

1. If $\partial_a(x) = \emptyset$ and $\partial_a(y) \neq \emptyset$, then $\delta(n_1, a) = (x\&\partial_a(y))z \cup \partial_a(z) = n_2 \cup n_3$
2. If $\partial_a(x) = \partial_a(y) = \emptyset$, then $\delta(n_1, a) = \partial_a(z)$

Proof.

1. If $\varepsilon \in L(x\&y)$, then

   $$\partial_a(x\&y)z = (\partial_a(x)\&y)z \cup (x\&\partial_a(y))z \cup \partial_a(z)$$

   If $\partial_a(x) = \emptyset$, then $\delta(n_1, a) = (x\&\partial_a(y))z \cup \partial_a(z) = n_2 \cup n_3$

2. If $\varepsilon \in L(x \,\&\, y)$ then

   $$\partial_a(x \,\&\, y)z = (\partial_a(x) \,\&\, y)\, z \cup (x \,\&\, \partial_a(y))\, z \cup \partial_a(z)$$

   $\partial_a(x) = \partial_a(y) = \emptyset$, implies that $\delta(n_1, a) = \partial_a(z)$ □

EXAMPLE 5.4.1. Given *PRE* $r = ((a \,\&\, b)c \,\&\, ab)$

States of the *NFA* are represented by natural numbers, and its corresponding *NFA* is shown in Figure 5.4.1.

$r = ((a \,\&\, b)c \,\&\, ab) = 0$

$\partial_a(r) = (bc \,\&\, ab) \cup ((a \,\&\, b)c \,\&\, b) = 1 \cup 3$

$\partial_b(r) = (ac \,\&\, ab) = 2$

$\partial_c(r) = \emptyset$

$\partial_a(bc \,\&\, ab) = bc \,\&\, b = 4$

$\partial_b(bc \,\&\, ab) = c \,\&\, ab = 5$

$\partial_a(ac \,\&\, ab) = (c \,\&\, ab) \cup (ac \,\&\, b) = 5 \cup 6$

$\partial_a((a \,\&\, b)c \,\&\, b) = bc \,\&\, b = 4$

$\partial_b((a \,\&\, b)c \,\&\, b) = (ac \,\&\, b) \cup ((a \,\&\, b)c) = 6 \cup 7$

$\partial_b(bc \,\&\, b) = (c \,\&\, b) \cup bc = 9 \cup 8$

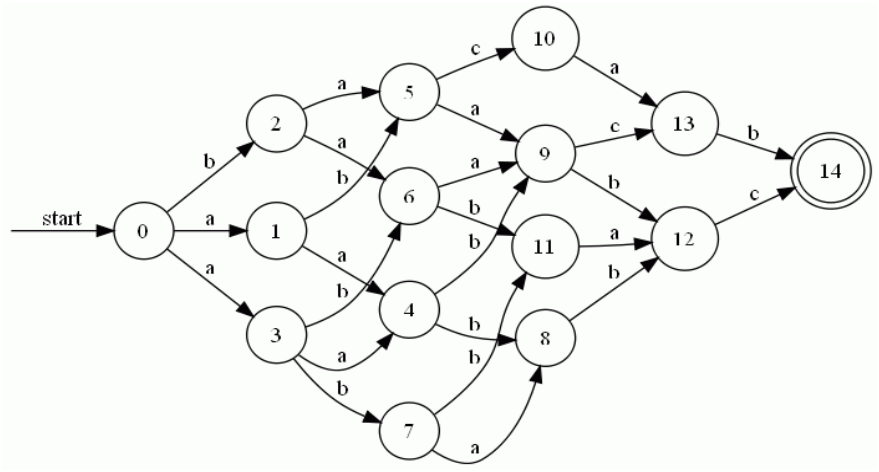


Figure 5.4.1: *NFA* for $r = ((a\&b)c\&ab)$ using partial derivatives

$\partial_a\,(c\,\&ab) \;=\; (c\,\&\,\partial_a\,(ab)) \;=\; (c\,\&\,b) \;=\; 9$

$\partial_c\,(c\,\&\,ab) \;=\; ab \;=\; 10$

$\partial_a\,(ac\,\&\,b) \;=\; (c\,\&\,b\,) = \; 9$

$\partial_b\,(ac\,\&b) \;=\; ac \;=\; 11$

$\partial_a\,((a\,\&\,b)c) \;=\; bc \;=\; 8$

$\partial_b\,((a\,\&\,b)c) \;=\; ac \;=\; 11$

$\partial_b\,(bc) \;=\; c \;=\; 12$

$\partial_b\,(c\,\&\,b) \;=\; c \;=\; 12$

$\partial_c\,(c\,\&\,b) \;=\; b \;=\; 13$

$\partial_a\,(ab) \;=\; b \;=\; 13$

$\partial_a\,(ac) \;=\; c \;=\; 12$

$\partial_b\,(b) \;=\; \varepsilon \;=\; 14$

$\partial_c\,(c) \;=\; \varepsilon \;=\; 14$

EXAMPLE 5.4.2. Given *PRE* $r \;=\; (a\&b)^*\&c^*$

Starting state $0 \;=\; (a\&b)^*\&c^*$

$\partial_a\,((a\,\&\,b)^*\&c^*) \;=\; (b(a\,\&\,b)^*)\,\&\,c^* \;=\; 1$

$\partial_b\ ((a\&b)^*\ \&\ c^*)\ =\ (a(a\ \&\ b)^*)\&c^*\ =\ 2$

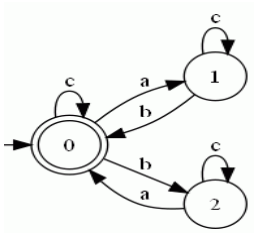


Figure 5.4.2: NFA for $r = (a\&b)^*\&c^*$ using partial derivatives

$\partial_c\ ((a\ \&\ b)^*\ \&\ c^*)\ =\ (a\ \&\ b)^*\ \&\ c^*\ =\ 0$

$\partial_a((b(a\ \&\ b)^*)\ \&\ c^*)\ =\ \emptyset$

$\partial_b((b(a\ \&\ b)^*)\ \&\ c^*)\ =\ (a\ \&\ b)^*\ \&\ c^*\ =\ 0$

$\partial_c((b(a\ \&\ b)^*)\ \&\ c^*)\ =\ (b(a\ \&\ b)^*)\ \&\ c^*\ =\ 1$

$\partial_a((a(a\ \&\ b)^*)\&c^*)\ =\ (a\ \&\ b)^*\ \&\ c^*\ =\ 0$

$\partial_c((a(a\ \&\ b)^*)\&c^*)\ =\ ((a(a\ \&\ b)^*)\ \&\ c^*)\ =\ 2$

LEMMA 5.4.4. *If PRE r consists of unit-level of shuffling. Then, using partial derivatives, the number of states obtained in the NFA is always less than* $(m+1)(m+2)/2$ *where m represents the number of instances of symbols in r.*

Proof. Using partial derivatives, worst case can occur if *PRE* $r = (a_1a_2...a_i)\&(b_1b_2...b_j)$ and $m = i+j$. Starting state $q_0 = (a_1a_2...a_i)\&(b_1b_2...b_j)$. We can represent the starting state by $(1,1)$ indicating that we are going to read a_1 and b_1 from state q_0.

$$\delta(q_0,a_1) = \partial_{a_1}(r) = (a_2...a_i)\&(b_1,b_2...b_j) = (2,1)$$
$$\delta(q_0,b_1) = \partial_{b_1}(r) = (a_1,a_2...a_i)\&(b_2...b_j) = (1,2)$$

Two new states are generated on reading of the first symbol. The number of new states generated is equal to the ordered integer partition of three, such that the maximum number is two, and the number of parts in the partition is two. $T_{i,j}$ denote the number of ordered integer partitions of j with the maximum number $(j-1)$ and

each partition consists of exactly i parts. If the sum of k numbers is n, then the range for picking k number is $[(n-k+1),...,1]$.

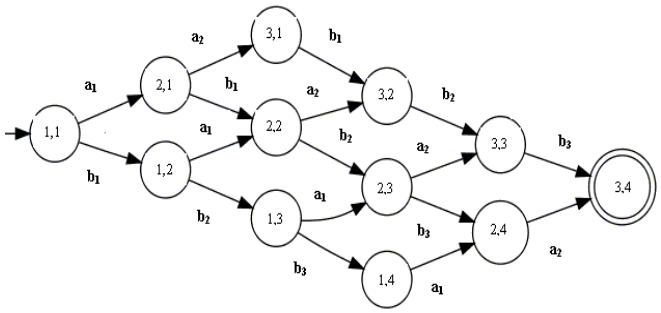


Figure 5.4.3: NFA for $r = (a_1a_2)\&(b_1b_2b_3)$

If $k = 2$ and $sum = 3$, then the range of each number is $[2,1]$. On reading of the second symbol, following additional states are generated:

$$\delta((2,1),a_2) = \partial_{a_2}(2,1) = (a_3,a_4,...,a_i)\&(b_1,b_2,...,b_j) = (3,1)$$
$$\delta((2,1),b_1) = \partial_{b_1}(2,1) = (a_2,a_3,...,a_i)\&(b_2,b_3,...,b_j) = (2,2)$$
$$\delta((1,2),a_1) = \partial_{a_1}(1,2) = (a_2,a_3,...,a_i)\&(b_2,b_3,...,b_j) = (2,2)$$
$$\delta((1,2),b_2) = \partial_{b_2}(1,2) = (a_1,a_2,...,a_i)\&(b_3,b_4,...,b_j) = (1,3)$$
$$T_{2,4} = 3$$

Repeating the above process, $T_{2,m+2} = (m+1)$.

Total number of states in the *NFA* is $\sum_{i=2}^{i=m+2}(T_{2,i}) = (m+1)(m+2)/2 = O(m^2)$.

In actual, number of states obtained is always less than $(m+1)(m+2)/2$. After reading of m^{th} symbol, we consider $(m+2)$ new states, but in actual there is only one final state. Suppose $i < j$ and after reading i^{th} symbol, $T_{2,(i+2)} = (i+1)$ new states are generated. We have no other options, but to read b_1 on state represented by $(i+1,1)$. Hence $(i+1)$ new states are generated for reading i^{th} to j^{th} symbol as shown in Figure 5.4.3. Number of new states generated on reading of $(j+1)^{th}$ and onwards symbols can be represented by $T_{2,(j+2+s)} = (i+1)-s$. The number of states generated in the worst case of a *NFA* is equal to

$ij+i+j+1$. Clearly $(ij+i+j+1) < ((m+1)(m+2)/2)$ where $m = i+j$. Hence, the number of states required for the *NFA* is always less than $(m+1)(m+2)/2$. □

For finding the number of states in multi-level shuffling, we consider the number of states in unit-level of shuffling is equal to $(m+1)(m+2)/2$.

LEMMA 5.4.5. *If PRE r consists of 2-level of shuffling. Using partial derivatives, the number of states obtained in the NFA is* $O(m^3)$.

Proof. Consider $m = i + j + k$ and *PRE* r is of the form of $(a_1,a_2,...,a_i)\&(b_1,b_2,...,b_j)\&(c_1,c_2,...,c_k)$. Starting state $q_0 = r$ and represented by $(1,1,1)$.

$$\delta(q_0,a_1) = \partial_{a_1}(r) = (a_2,...,a_i)\&(b_1,...,b_j)\&(c_1,...,c_k) = (2,1,1)$$
$$\delta(q_0,b_1) = \partial_{b_1}(r) = (a_1,...,a_i)\&(b_2,...,b_j)\&(c_1,...,c_k) = (1,2,1)$$
$$\delta(q_0,c_1) = \partial_{c_1}(r) = (a_1,...,a_i)\&(b_1,...,b_j)\&(c_2,...,c_k) = (1,1,2)$$

Clearly range of picking three numbers is $[4-3+1,...,1]$.

$$T_{3,4} = 3 = T_{2,2} + T_{2,3}$$

On reading second symbol, the number of new state generated is equal to

$$T_{3,5} = T_{2,4} + T_{2,3} + T_{2,2}$$

Repeating the above process, if the sum of three numbers is n then

$$T_{3,n} = T_{2,n-1} + T_{2,n-2} + T_{2,2} = (n-2)(n-1)/2$$

Number of states in the *NFA* is $\sum_{i=3}^{i=m+3}(T_{3,i}) = (m^3/6 + m^2 + 11m/6 + 1) = O(m^3)$□

THEOREM 5.4.1. *If PRE r consists of k-level of shuffling. Using partial derivatives, the number of states obtained in the NFA is* $O(m^{k+1})$.

Proof. If r consists of k-level of shuffling, the starting state will be represented by k-parts whose sum is k. If the sum of k numbers is n, then range of the k numbers is [*n-k+1*,...,*1*].

For $n > 4$, $T_{4,n} = T_{3,n-1} + T_{3,n-2} + ... + T_{3,3}$ and the number of new states after reading each symbol is of the form of n^3.

If $n = 4$, $T_{4,4} = 1$

Similarly,

For $n > k, T_{k,n} = T_{k-1,n-1} + T_{k-1,n-2} + ... + T_{k-1,k-1}$ and is a polynomial of the form of $O(n^k)$.

If $n = k, T_{k,n} = 1$

By using the Bernoulli's formula [26], for summing the series of natural number, we obtain a polynomial of degree $(k+1)$. Hence the number of states obtained in the *NFA* is $O(m^{k+1})$. □

5.5 CONCLUSION

In this chapter, the worst case state complexity of *PREs* to *NFAs* is explored using the partial derivatives. The number of states generated using this approach in the worst case is $O(m^{k+1})$, which is a significant improvement as compared to other existing approaches till date.

Chapter 6

AN ALGORITHM FOR METAMORPHOSIS OF PARALLEL REGULAR EXPRESSION TO RIGHT LINEAR GRAMMAR BASED ON THE FOLLOW-POSITION OF SYMBOLS

6.1 INTRODUCTION

This chapter proposes an algorithm for metamorphosis of parallel regular expression (*PRE*) to right linear grammar, using the concept of follow-position of the symbols. The metamorphosis of *PRE* to regular grammar gives a prescript of reduction in concurrency.

In the neoteric years, studies of regular expressions with shuffle operator have attracted significant attention due to its theoretical and practical applications in the field of XML schema language, concurrent system, modeling of process algebra, multipoint communication and critical section problem etc. [6, 35-36, 41, 56]. Therefore, conversion of the *PRE* to regular grammar attain significant importance.

The following section describes the preliminaries and then the proposed algorithm is described followed by numerical examples.

6.2 PRELIMINARIES

DEFINITION 6.2.1. A grammar [2] $G = (N, \Sigma, S, P)$ consists of an alphabet of non-terminals N, an alphabet of terminals Σ, a starting symbol $S \in N$ and a finite set of production $P \subseteq (N \cup \Sigma)^* \times (N \cup \Sigma)^*$.

DEFINITION 6.2.2. A grammar $G = (N, \Sigma, S, P)$ is said to be right linear if the production $P \subseteq N \times (\Sigma \cup (\Sigma \circ N) \cup \{\varepsilon\})$.

DEFINITION 6.2.3. A grammar $G = (N, \Sigma, S, P)$ is said to be left linear if the production $P \subseteq N \times (\Sigma \cup (N \circ \Sigma) \cup \{\varepsilon\})$.

DEFINITION 6.2.4. A grammar $G = (N, \Sigma, S, P)$ is said to be regular grammar if it is either left linear or right linear.

EXAMPLE 6.2.1. Given the regular expression $r = ab(cb|bb)^*$.

1. Right linear grammar

   $S \rightarrow abA$

   $A \rightarrow cbA|bbA|\varepsilon$

2. Left linear grammar

   $S \rightarrow Scb|Sbb|ab$

DEFINITION 6.2.5. A production $p \in P$ is null production [2] if $p \in (N \times \{\varepsilon\})$.

6.3 PROPOSED ALGORITHM

The following section highlights the proposed algorithm and as per the literature review no such algorithm for conversion of *PRE* to regular grammar has been proposed. Algorithm 6.3.1, converts *PRE* to a right linear grammar G such that $L(G) = L(r)$. This metamorphosis consists of following six steps:

1. **Augmented *PRE*:** Add # at the cessation of *PRE* r. If $(r_i \& r_j) \in r$, add # after each r_j but immediately before the next symbol $a \in \Sigma$ or $a =$ #. Addition of # enlighten the end of extant shuffle operation. Assign positions to the symbols of the augmented *PRE*. Position of the leftmost symbol is one and increases towards the right side of the symbols. Generate a non-terminal corresponding to each position. Initially, set $N = \{S, S_1, S_2, ..., S_n\}$. Non-terminal S_i suggests that we are going to process i^{th} symbol.

2. ***followpos* Calculation:** Procedure *follow* partition the set N into two sets. First set consists of the non-terminals formed from the positions of the symbols besmeared in the shuffling. The second set consists of non-terminals

whose corresponding symbol is not besmeared in shuffling. Using the procedure *follow*, *followpos* of each position assigned to the symbols are determined. Set *union_left* consists of minimum position of each r_i where $(r_i|r_j) \in r$. Set *union_right* consists of minimum position of each r_j where $(r_i|r_j) \in r$. Non-terminal S_{IJK} denotes a non-terminal where $I = firstpos(r_m)$, $J = firstpos(r_n)$, K is the position of # immediately after r_n and $(r_m \& r_n) \in r$. Non-terminal S_{IJK} indicates that we are going to process a symbol from a position $i \in I$ or $j \in J$.

At shuffle operator:

$$firstpos(\&) = firstpos(r_m) \cup firstpos(r_n)$$

$$lastpos(\&) = lastpos(r_m) \cup lastpos(r_n)$$

$$nullable(\&) = nullable(r_m) \wedge nullable(r_n)$$

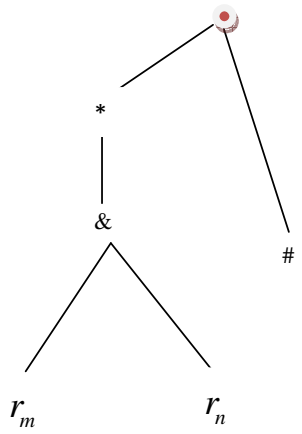


Figure 6.3.1: Closure of shuffle operator

If $i \in lastpos(r_m)$, add E to $followpos(i)$. Similarly, if $j \in lastpos(r_n)$, add E to $followpos(i)$. *Followpos* of the symbols involved in the sub expressions r_m and

r_n can be determined using the well known methodology [20]. Consider S_{IJK} is the starting non-terminal of $r_m\&r_n$.

Figure 6.3.1, shows that if $(r_m\&r_n)^* \in r'$, add a production rule $S_K \rightarrow S_{IJK}$.

If $\varepsilon \in r_m$, add a production rule $S_{IJK} \rightarrow S_{EJK}$.

Similarly, if $\varepsilon \in r_n$, add a production rule $S_{IJK} \rightarrow S_{IEK}$.

If $\varepsilon \in (r_m\&r_n)$, add a production rule $S_{IJK} \rightarrow S_{EEK}$.

3. **Productions of the positions not involved in shuffling:** Set $first(r')$ gives the positions of the first symbol generated from r'. Consider $first(r') = \{i,j,k\}$, add production $S \rightarrow S_i|S_j|S_k$. It means, we can continue to one of the non-terminal $\{S_i, S_j, S_k\}$ from the starting non-terminal S. Add production rule corresponding to each position i using the following rules:

   For each $(j \in followpos(i))$

   If $(a_i \in \Sigma)$

   $\langle S_i \rangle \rightarrow a_i \langle S_j \rangle$

   Else

   $\langle S_i \rangle \rightarrow \langle S_j \rangle$

   End If

   End of Loop

4. **Shuffle Productions:** Consider $(r_m\&r_m) \in r'$. Traverse r' in preorder. Smallest position symbol of r_m occurs after the shuffle operator. Starting non-terminal S_{IJK} can be found using the smallest position of r_m. Productions of sub expression $(r_m\&r_m)$ can be generated using the procedure shuffle-productions(G, r', S_X). $(K-1)$ gives the highest position of the symbol of sub expression $(r_m\&r_m)$. *lower_left* and *lower_right* provides the lowest position

of left and right sub expression besmeared in the union operator. In the union operator, one of the paths is chosen from the two possible paths. If the *followpos* of a position contains a value less than or equal to itself, it means a Kleene closure operator occurs. Whenever a symbol at a position occurs in Kleene closure, wariness should be taken to dislodge smaller values while considering the larger values.

5. **Removal of S_n non-terminal:** A non-terminal $S_n \in N$ produces a null production if $S_n \rightarrow \varepsilon$. Null production of S_n can be detached by simply substituting ε in place of S_n in the right side of all productions.

6. **Removal of terminating and unit productions:** A non-terminal $S_i \in N$ produces a terminating production if $p_i : N \times \{T^*\}$. Terminating production can be removed by simply substituting the right hand side of production in place of S_i in the right side of all productions. A non-terminal $S_i \in N$ is in unit production if $p_i : N \times \{N\}$. Unit production can be removed by simply substituting the right hand side non-terminal in place of S_i in all productions.

Notations used in the algorithm: The following notations are used in the algorithm:
current: It represents current symbol. It can be one of the symbols from Σ, an operator or # symbol.
pos: *pos* represents the position of a symbol.
***left*:** It represents the left child of a node in the syntax tree. Single child of a node is also represented by *left*.
***right*:** It represents the right child of a node in the syntax tree.

Algorithm 6.3.1: *PRE_RLG*(r)

Input: *PRE r*
Output: Regular grammar G such that $L(G)$=$L(r)$

Augmented *PRE* r'$\leftarrow r\#$
if $((r_i \& r_j) \in r)$ **then**
    Add # after r_j but immediately before the next symbol a where $((a = \#) \vee (a \in \Sigma))$
end
Assign positions to the symbols from Σ and $\# \in r'$.
$N \leftarrow \{S, S_1, S_2, ..., S_n\}$ and $N' \leftarrow \{\}$
Follow(r', N)
for $(\forall i \in first(r'))$ **do**
    if $(S_i \in N)$ **then**
        $P \leftarrow P \cup \{\langle S\rangle \rightarrow \langle S_i\rangle\}$
    end
end
for $(\forall S_i \in N)$ **do**
    if $(S_i \neq S)$ **then**
        for $(\forall j \in followpos(i))$ **do**
            if $(a_i \in \Sigma)$ **then**
                if $(S_j \in N)$ **then**
                    $P \leftarrow P \cup \{\langle S_i\rangle \rightarrow a_i\langle S_j\rangle\}$
                else
                    Choose X such that $((j \in X) \wedge (S_X \in N'))$
                    $P \leftarrow P \cup \{\langle S_i\rangle \rightarrow a_i\langle S_X\rangle\}$
                end
            else
                $P \leftarrow P \cup \{\langle S_i\rangle \rightarrow \langle S_j\rangle\}$
            end
        end
    end
end
$P \leftarrow P \cup \{\langle S_n\rangle \rightarrow \langle \varepsilon\rangle\}$
$pos \leftarrow 0$
$shuffle \leftarrow false$

▷ **Continued**...

Algorithm 6.3.1: *PRE_RLG(r)* **Continued...**

```
31 for ((∀current ∈ Preorder(r')) ∧ (Scan Preorder(r') from left to right)) do
32     if (current ∈ Σ) then
33         pos ← pos + 1
34         if (shuffle=true) then
35             shuffle ← false
36             Choose X such that (pos ∈ X) ∧ (S_X ∈ N') ∧ (E ∉ X)
37             if (pos ∈ firstpos(r')) then
38                 P ← P ∪ {⟨S⟩ → ⟨S_X⟩}
39             end
40             shuffle-production(r', S_X)
41         end
42     end
43     if (current = &) then
44         shuffle←true
45     end
46     Eliminate_#-production(G)
47     Eliminate_terminating-production(G)
48     Eliminate_unit-production(G)
49 end
```

Example 6.3.1 Consider *PRE* $r = e(ab\&cd)^*f$

Using algorithm 6.3.1, *PRE* $r' = e_1(a_2b_3\&c_4d_5)^*\#_6f_7\#_8$

$N = \{S, S_1, ..., S_8\}$

$N' = \{\}$

Using algorithm 6.3.2

$$union\_left = union\_right = \{\}$$

$$followpos(1) = \{2,4,6\}$$

$$followpos(2) = \{3\}$$

$$followpos(3) = followpos(5) = \{E\}$$

$$followpos(4) = \{5\}$$

$$followpos(6) = \{7\}$$

$$followpos(7) = \{8\}$$

$$followpos(8) = \{\}$$
$$union\_left = union\_right = \{\}$$
$$N = \{S, S_1, S_6, S_7, S_8\}$$
$$N' = \{S_{246}\}$$

Starting and ending non-terminals of $(ab\&cd)$ are S_{246} and S_6 respectively. Sub expression $(ab\&cd)$ can occur zero or more times. For second and onward iteration, we add a production rule $\langle S_6 \rangle \rightarrow \langle S_{246} \rangle$ to P.

Using algorithm 6.3.1, add productions corresponding to the positions whose symbol are not involved in shuffling.

$$firstpos(r') = \{1\}, P \leftarrow P \cup \{\langle S \rangle \rightarrow \langle S_1 \rangle\}$$
$$followpos(1) = \{2,4,6\}, P \leftarrow P \cup \{\langle S_1 \rangle \rightarrow e\langle S_6 \rangle | e\langle S_{246} \rangle\}$$
$$followpos(6) = \{7\}, P \leftarrow P \cup \{\langle S_6 \rangle \rightarrow \langle S_7 \rangle\}$$
$$followpos(7) = \{8\}, P \leftarrow P \cup \{\langle S_7 \rangle \rightarrow f\langle S_8 \rangle\}$$
$$P \leftarrow P \cup \{\langle S_8 \rangle \rightarrow \varepsilon\}$$

Preorder of $r' = ...e^*\&.ab.cd\#f\#$

$pos = 2$

Procedure shuffle-productions(G, r', S_{246}) generates the productions of the position involved in shuffling. □

Various rules for calculating *followpos*, *union_left*, *union_right* and productions are given in Table 6.3.1. These rules in conjunction with the rules proposed in direct conversion method [20] are used in algorithm 6.3.2.

In the shuffle-production procedure, various productions and non-terminals are generated that occurs due to shuffle operator. Starting positions of the Kleene closure are found using step 2 of the algorithm 6.3.3.

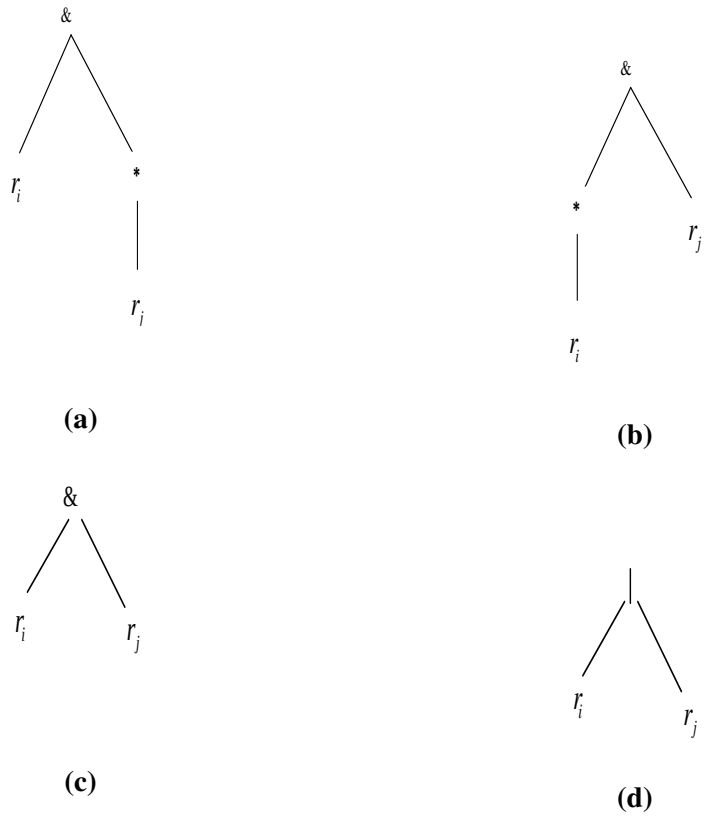


Figure 6.3.2: Syntax tree of the sub expression

Table 6.3.1: Rules for determining various values and productions occurs due to shuffle operator

Syntax tree	Current node	Production rules
Figure 6.3.2 (c)	Shuffle operator	$firstpos(\&) = firstpos(r_i) \cup firstpos(r_j)$ $lastpos(\&) = lastpos(r_i) \cup lastpos(r_j)$ $nullable(\&) = nullable(r_i) \wedge nullable(r_j)$
Figure 6.3.2 (d)	Union operator	$union\_left \leftarrow union\_left \cup \mathbf{Min}(firstpos(r_i))$ $union\_right \leftarrow union\_right \cup \mathbf{Min}(firstpos(r_j))$

Table 6.3.1: Rules for determining various values and productions occurs due to shuffle operator **Continued...**

Syntax tree	Current node	Production rules
Figure 6.3.1	Kleene closure	$\#pos = position(\#)$ $X \leftarrow firstpos(r_i) \cup firstpos(r_j) \cup \#pos$ $\langle S_{\#pos} \rangle \rightarrow \langle S_X \rangle$
Figure 6.3.2 (a)	Shuffle operator	$\#pos = position(\#)$ $X \leftarrow firstpos(r_i) \cup firstpos(r_j) \cup \#pos$ $Y \leftarrow firstpos(r_i) \cup E \cup \#pos$ $\langle S_X \rangle \rightarrow \langle S_Y \rangle$ E indicate that we are taking ε from r_j
Figure 6.3.2 (b)	Shuffle operator	$\#pos = position(\#)$ $X \leftarrow firstpos(r_i) \cup firstpos(r_j) \cup \#pos$ $Y \leftarrow E \cup firstpos(r_j) \cup \#pos$ $\langle S_X \rangle \rightarrow \langle S_Y \rangle$ E indicate that we are taking ε from r_i

Algorithm 6.3.2: *Follow*(r', N)

Input: *Augmented PRE r'*
Output: *firstpos*, *lastpos*, *followpos*, N and N'

1 Draw a syntax tree for r' such that the symbols from Σ and # appear as the leaf node.
2 $union\_left \leftarrow union\_right \leftarrow \{\}$
3 Traverse the syntax tree in post-order
4 **if** $((current \in \Sigma) \vee (current = \#))$ **then**
5 $\quad firstpos(current) \leftarrow position(current)$
6 $\quad lastpos(current) \leftarrow position(current)$
7 **end**

Algorithm 6.3.2: *Follow*(r', N) **Continued**...

```
8  if (current = &) then
9      firstpos(current) ← firstpos(left) ∪ firstpos(right)
10     lastpos(current) ← lastpos(left) ∪ lastpos(right)
11     nullable(current) ← nullable(left) ∧ nullable(right)
12     for (∀i ∈ lastpos(current)) do
13         followpos(i) ← followpos(i) ∪ {E}
14     end
15     i_1 ← Min(firstpos(left))
16     i_2 ← Max(lastpos(right))
17     for (i ← i_1; i ≤ i_2; i ← i + 1) do
18         N ← N − S_i
19         i ← i + 1
20     end
21     X ← firstpos(current) ∪ (i_2 + 1)
22     N' ← N' ∪ S_X
23     if (nullable(current) = true) then
24         P ← P ∪ {⟨S_X⟩ → ε}
25     end
26     if (nullable(left) = true) then
27         Y ← E ∪ firstpos(right) ∪ (i_2 + 1)
28         N' ← N' ∪ S_Y
29         P ← P ∪ {⟨S_X⟩ → ⟨S_Y⟩}
30     end
31     if (nullable(right) = true) then
32         Y ← E ∪ firstpos(left) ∪ (i_2 + 1)
33         N' ← N' ∪ S_Y
34         P ← P ∪ {⟨S_X⟩ → ⟨S_Y⟩}
35     end
36 end
37 if (current = |) then
38     firstpos(current) ← firstpos(left) ∪ firstpos(right)
39     lastpos(current) ← lastpos(left) ∪ lastpos(right)
40     if ((nullable(left) = true) ∨ (nullable(right) = true)) then
41         nullable(current) ← true
42     end
43     union_left ← union_left ∪ Min(firstpos(left))
44     union_right ← union_right ∪ Min(firstpos(right))
45 end
```

Algorithm 6.3.2: *Follow*(r', N) **Continued**...

```
46 if (current = ∗) then
47     firstpos(current) ← firstpos(left)
48     lastpos(current) ← lastpos(left)
49     nullable(current) ← true
50     if (left ≠ &) then
51         for (∀j ∈ lastpos(current)) do
52             followpos(j) ← followpos(j) ∪ firstpos(current)
53         end
54     else
55         Let S_X be the recently added non-terminal to N'
56         #pos ← Max(X)
57         P ← P ∪ {⟨S_#pos⟩ → ⟨S_X⟩}
58     end
59 end
60 if (current = .) then
61     firstpos(current) ← firstpos(left)
62     lastpos(current) ← lastpos(right)
63     if (nullable(left) = true) then
64         firstpos(current) ← firstpos(current) ∪ firstpos(right)
65     end
66     if (nullable(right) = true) then
67         lastpos(current) ← lastpos(current) ∪ lastpos(left)
68     end
69     if ((nullable(left) = true) ∧ (nullable(right) = true)) then
70         nullable(current) ← true
71     end
72     if (left ≠ &) then
73         for (∀j ∈ lastpos(left)) do
74             if (E ∉ followpos(j)) then
75                 followpos(j) ← followpos(j) ∪ firstpos(right)
76             end
77         end
78     end
79 end
```

Example 6.3.2 Apply $shuffle - productions(r', S_X)$ to Example 6.3.1.

For non-terminal S_{246}

$closure\_start = \{\}$

$i = 2$

$i \notin union\_left$ and $i \notin union\_right$

$P \leftarrow P \cup \{\langle S_{246} \rangle \rightarrow a\langle S_{346} \rangle\}$

$NT' = \{S_{246}, S_{346}\}$

$i = 4$

$i \notin union\_left$ and $i \notin union\_right$

$P \leftarrow P \cup \{\langle S_{246} \rangle \rightarrow c\langle S_{256} \rangle\}$

$NT' = \{S_{246}, S_{346}, S_{256}\}$

For non-terminal S_{346}

$i = 3$

$i \notin union\_left$ and $i \notin union\_right$

$P \leftarrow P \cup \{\langle S_{346} \rangle \rightarrow b\langle S_{E46} \rangle\}$

$NT' = \{S_{246}, S_{346}, S_{256}, S_{E46}\}$

$i = 4$

$i \notin union\_left$ and $i \notin union\_right$

$P \leftarrow P \cup \{\langle S_{346} \rangle \rightarrow c\langle S_{356} \rangle\}$

$NT' = \{S_{246}, S_{346}, S_{256}, S_{E46}, S_{356}\}$

For non-terminal S_{256}

$i = 2$

$P \leftarrow P \cup \{\langle S_{256} \rangle \rightarrow a\langle S_{356} \rangle\}$

$NT' = \{S_{246}, S_{346}, S_{256}, S_{E46}, S_{356}\}$

$i = 5$

$P \leftarrow P \cup \{\langle S_{256} \rangle \rightarrow d\langle S_{2E6} \rangle\}$

$NT' = \{S_{246}, S_{346}, S_{256}, S_{E46}, S_{356}, S_{2E6}\}$

For non-terminal S_{E46}

$i = 4$

$$P \leftarrow P \cup \{\langle S_{E46}\rangle \rightarrow c\langle S_{E56}\rangle\}$$

$$NT' = \{S_{246}, S_{346}, S_{256}, S_{E46}, S_{356}, S_{2E6}, S_{E56}\}$$

For non-terminal S_{356}

$i = 3$

$$P \leftarrow P \cup \{\langle S_{356}\rangle \rightarrow b\langle S_{E56}\rangle\}$$

$$NT' = \{S_{246}, S_{346}, S_{256}, S_{E46}, S_{356}, S_{2E6}, S_{E56}\}$$

$i = 5$

$$P \leftarrow P \cup \{\langle S_{356}\rangle \rightarrow d\langle S_{3E6}\rangle\}$$

$$NT' = \{S_{246}, S_{346}, S_{256}, S_{E46}, S_{356}, S_{2E6}, S_{E56}, S_{3E6}\}$$

For non-terminal S_{2E6}

$i = 2$

$$P \leftarrow P \cup \{\langle S_{2E6}\rangle \rightarrow a\langle S_{3E6}\rangle\}$$

$$NT' = \{S_{246}, S_{346}, S_{256}, S_{E46}, S_{356}, S_{2E6}, S_{E56}, S_{3E6}\}$$

For non-terminal S_{E56}

$i = 5$

$$P \leftarrow P \cup \{\langle S_{E56}\rangle \rightarrow d\langle S_{EE6}\rangle\}$$

$$NT' = \{S_{246}, S_{346}, S_{256}, S_{E46}, S_{356}, S_{2E6}, S_{E56}, S_{3E6}, S_{EE6}\}$$

For non-terminal S_{3E6}

$i = 3$

$$P \leftarrow P \cup \{\langle S_{3E6}\rangle \rightarrow b\langle S_{EE6}\rangle\}$$

$$NT' = \{S_{246}, S_{346}, S_{256}, S_{E46}, S_{356}, S_{2E6}, S_{E56}, S_{3E6}, S_{EE6}\}$$

Production ***P***

$$\langle S\rangle \rightarrow \langle S_1\rangle$$

$$\langle S_1\rangle \rightarrow e\langle S_6\rangle | e\langle S_{246}\rangle$$

$$\langle S_6\rangle \rightarrow \langle S_7\rangle | \langle S_{246}\rangle$$

$$\langle S_7\rangle \rightarrow f\langle S_8\rangle$$

$$\langle S_8 \rangle \rightarrow \langle \varepsilon \rangle$$

$$\langle S_{246} \rangle \rightarrow a\langle S_{346} \rangle | c\langle S_{256} \rangle$$

$$\langle S_{346} \rangle \rightarrow b\langle S_{E46} \rangle | c\langle S_{356} \rangle$$

$$\langle S_{256} \rangle \rightarrow a\langle S_{356} \rangle | d\langle S_{2E6} \rangle$$

$$\langle S_{E46} \rangle \rightarrow c\langle S_{E56} \rangle$$

$$\langle S_{356} \rangle \rightarrow b\langle S_{E56} \rangle | d\langle S_{3E6} \rangle$$

$$\langle S_{2E6} \rangle \rightarrow a\langle S_{3E6} \rangle$$

$$\langle S_{E56} \rangle \rightarrow d\langle S_6 \rangle$$

$$\langle S_{3E6} \rangle \rightarrow b\langle S_6 \rangle$$

Algorithm 6.3.3: shuffle-production(r', S_X)

Input: *Augmented PRE r'.*
Output: Set of non-terminal *NT'* and *Productions*.

```
1  NT' ← S_X
2  closure_start ← {}
3  for (i ← Min(X); i ≤ Max(X) − 1; i = i + 1) do
4      closure ← false
5      for (j ∈ followpos(i)) do
6          if ((j ≤ i) ∧ (j ≠ E)) then
7              closure_start ← closure_start ∪ j
8              closure ← true
9          end
10     end
11     if (closure = true) then
12         for (∀k ∈ followpos(i)) do
13             if ((k > i) ∧ (k ≠ E)) then
14                 closure_start ← closure_start ∪ k
15             end
16         end
17     end
18 end
```

Algorithm 6.3.3: shuffle-production(r', S_X) **Continued...**

```
19 for (∀S_X ∈ NT') do
20     temp ← X
21     j ← {}
22     for (∀i ∈ temp) do
23         X' ← X
24         if (a_i ∈ Σ) then
25             if (i ∈ union_left) then
26                 Choose{j|((j ∈ union_right) ∧ (index[i] = index[j]))}
27                 X' = X' − j
28             else
29                 if (i ∈ union_right) then
30                     Choose{j|((j ∈ union_left) ∧ (index[i] = index[j]))}
31                     X' = X' − j
32                 end
33             end
34             if (j ∈ closure_start) then
35                 for (∀k ∈ closure_start) do
36                     if ((k > j) ∧ (j,k ∈ followpos(k − 1))) then
37                         X' ← X' − followpos(k − 1)
38                     end
39                 end
40             end
41             temp1 ← X'
42             if (i ∈ closure_start) then
43                 for (∀k ∈ closure_start) do
44                     if ((k > i) ∧ (i,k ∈ followpos(k − 1))) then
45                         X' ← X' − followpos(k − 1)
46                     end
47                 end
48             end
49             temp1 ← temp1 − X' + i
50             for (i ∈ temp1) do
51                 for (∀j ∈ followpos(i)) do
52                     X'' ← X' − i
53                     X'' ← X'' + j
54                     P ← P ∪ {S_X → a_i S_X''}
55                     NT' ← NT' ∪ S_X''
56                 end
57             end
58             temp ← temp − temp1
59         end
60     end
61 end
```

Algorithm 6.3.3: shuffle-production(r', S_X) **Continued...**

62 Find $S_X | (E \in X) \wedge (\#_{pos} \in X) \wedge (a_{pos} \notin X)$
63 $X' \leftarrow X' - \{E\}$
64 $P \leftarrow P \cup \{S_X \rightarrow S_{X'}\}$
65 $N \leftarrow N \cup NT'$

On applying algorithm 6.3.4, the productions $\langle S_7 \rangle \rightarrow f$ and $\langle S_8 \rangle \rightarrow \varepsilon$ are removed.

Procedure *Eliminate_#-production*(G) eliminates the non-terminal S_n and its corresponding productions.

Algorithm 6.3.4: Eliminate_#production(G)

Input: Regular grammar $G = (N, T, S, P)$
Output: Regular grammar G' such that $L(G') = L(G)$

```
1 for (∀p ∈ P) do
2     if (S_n occurs in the right side of production p) then
3         Replace ⟨S_n⟩ → ε
4         N ← N − S_n
5         P ← P − {⟨S_n⟩ → ε}
6     end
7 end
```

Procedure *Eliminate_terminating−production*(G) removes the non-terminals which only produces terminal strings. Procedure *Eliminate_unit-production*(G) deletes the non-terminals which only produces unit productions.

Algorithm 6.3.5: Eliminate_terminating-production(G)

Input: Regular grammar $G = (N, T, S, P)$
Output: Regular grammar G' such that $L(G') = L(G)$

```
1 for (∀p ∈ P) do
2     if (p : ⟨S_i⟩ → w|w ∈ Σ*) then
3         Replace ⟨S_i⟩ → w in all productions having S_i in the right hand side.
4         P = P − {⟨S_i⟩ → w}
5         N ← N − S_i
6     end
7 end
```

Example 6.3.3 On applying the algorithm 6.3.5 and 6.3.6 to the production P of example 6.3.2.

Production *P*

$$\langle S\rangle \rightarrow e\langle S_6\rangle | e\langle S_{246}\rangle$$
$$\langle S_6\rangle \rightarrow f | \langle S_{246}\rangle$$
$$\langle S_{246}\rangle \rightarrow a\langle S_{346}\rangle | c\langle S_{256}\rangle$$
$$\langle S_{346}\rangle \rightarrow b\langle S_{E46}\rangle | c\langle S_{356}\rangle$$
$$\langle S_{256}\rangle \rightarrow a\langle S_{356}\rangle | d\langle S_{2E6}\rangle$$
$$\langle S_{E46}\rangle \rightarrow c\langle S_{E56}\rangle$$
$$\langle S_{356}\rangle \rightarrow b\langle S_{E56}\rangle | d\langle S_{3E6}\rangle$$
$$\langle S_{2E6}\rangle \rightarrow a\langle S_{3E6}\rangle$$
$$\langle S_{E56}\rangle \rightarrow d\langle S_6\rangle$$
$$\langle S_{3E6}\rangle \rightarrow b\langle S_6\rangle$$

Algorithm 6.3.6: Eliminate_unit-production(G)

Input: Regular grammar $G = (N, T, S, P)$
Output: Regular grammar G' such that $L(G') = L(G)$

```
for ∀p ∈ P do
    if (⟨S_i⟩ → ⟨S_j⟩) then
        Replace S_i → S_j in all productions having S_i in the right hand side.
        N ← N − S_i
    end
end
```

6.4. NUMERICAL EXAMPLES

In this section, two numerical examples are given for illustrating the metamorphosis of *PRE* to regular grammar.

Example 6.4.1 Let $r = a((b^*c|d)\&e)f$

$r' = a_1((b_2^*c_3|d_4)\&e_5)\#_6 f_7 \#_8$

$N = \{S, S_1, S_2, ..., S_8\}$

$N' = \{\}$

$union\_left = union\_right = \{\}$

$followpos(1) = \{2,3,4,5\}$

$followpos(2) = \{2,3\}$

$followpos(3) = followpos(4) = followpos(5) = \{E\}$

$followpos(6) = \{7\}$

$followpos(7) = \{8\}$

$followpos(8) = \{\}$

$union\_left = \{2\}$

$union\_right = \{4\}$

$N = \{S, S_1, S_6, S_7, S_8\}$

$N' = \{S_{23456}\}$

$firstpos(r') = \{1\}$

$P \leftarrow P \cup \{\langle S \rangle \rightarrow \langle S_1 \rangle\}$

$P \leftarrow P \cup \{\langle S_1 \rangle \rightarrow a\langle S_{23456} \rangle\}$

$P \leftarrow P \cup \{\langle S_6 \rangle \rightarrow \langle S_7 \rangle\}$

$P \leftarrow P \cup \langle \{S_7\rangle \rightarrow f\langle S_8 \rangle\}$

$P \leftarrow P \cup \{\langle S_8 \rangle \rightarrow \varepsilon\}$

$pos \leftarrow 2$

$shuffle - productions(G, r', S_{23456})$

For non-terminal S_{23456}

$union\_left = \{2\}, union\_right = \{4\}, closure\_start = \{2,3\}$

$temp = X' = X = \{2,3,4,5,6\}$

$i = 2$

$i \in union\_left$

$j = 4, X' = \{2,3,5,6\}$

$temp1 = \{2,3,5,6\}$

$i \in closure\_start, k = 3$ and $i, k \in followpos(k-1)$

$X' = \{2,3,5,6\} - \{2,3\} = \{5,6\}$

$temp1 = temp1 - X' = \{2,3,5,6\} - \{5,6\} = \{2,3\}$

$i = 2$

$P \leftarrow P \cup \{\langle S_{23456} \rangle \rightarrow b\langle S_{256} \rangle\}$

$P \leftarrow P \cup \{\langle S_{23456} \rangle \rightarrow b\langle S_{356} \rangle\}$

$i = 3$

$P \leftarrow P \cup \{\langle S_{23456} \rangle \rightarrow c\langle S_{E56} \rangle\}$

$NT' = \{S_{23456}, S_{256}, S_{356}, S_{E56}\}$

$temp = \{2,3,4,5,6\} - \{2,3\} = \{4,5,6\}$

$i = 4$

$X' = X = \{2,3,4,5,6\}$

$union\_left = \{2\}, union\_right = \{4\}, closure\_observe = \{2,3\}$

$i \in union\_right, X' = \{3,4,5,6\}$

$j \in closure\_start$

$X' = \{3,4,5,6\} - \{2,3\} = \{4,5,6\}$

$temp1 = \{4\}$

$P \leftarrow P \cup \{\langle S_{23456} \rangle \rightarrow d\langle S_{E56} \rangle\}$

$NT' = \{S_{23456}, S_{256}, S_{356}, S_{E56}\}$

$temp = \{4,5,6\} - \{4\} = \{5,6\}$

$i = 5$

$X' = X = \{2,3,4,5,6\}$

$union\_left = \{2\}, union\_right = \{4\}, closure - observe = \{2,3\}$

$temp1 = \{5\}$

$P \leftarrow P \cup \{\langle S_{23456} \rangle \rightarrow e\langle S_{234E6} \rangle\}$

$NT' = \{S_{23456}, S_{256}, S_{356}, S_{E56}, S_{234E6}\}$

For non-terminal S_{256}

$i = 2$

$temp = X' = X = \{2,5,6\}$

$union\_left = \{2\}, union\_right = \{4\}, closure - observe = \{2,3\}$

$temp1 = X' = X' - \{4\} = \{2,5,6\}$

$temp1 = temp1 - X' + i = i = \{2\}$

$P \leftarrow P \cup \{\langle S_{256} \rangle \rightarrow b\langle S_{256} \rangle\}$

$P \leftarrow P \cup \{\langle S_{256} \rangle \rightarrow b\langle S_{356} \rangle\}$

$NT' = \{S_{23456}, S_{256}, S_{356}, S_{E56}, S_{234E6}\}$

$i = 5$

$P \leftarrow P \cup \{\langle S_{256} \rangle \rightarrow e\langle S_{2E6} \rangle\}$

$NT' = \{S_{23456}, S_{256}, S_{356}, S_{E56}, S_{234E6}, S_{2E6}\}$

For non-terminal S_{356}

$i = 3$

$temp = X' = X = \{3,5,6\}$

$union\_left = \{2\}, union\_right = \{4\}, closure - observe = \{2,3\}$

$P \leftarrow P \cup \{\langle S_{356} \rangle \rightarrow c\langle S_{E56} \rangle\}$

$NT' = \{S_{23456}, S_{256}, S_{356}, S_{E56}, S_{234E6}, S_{2E6}\}$

$i = 5$

$P \leftarrow P \cup \{\langle S_{356} \rangle \rightarrow e\langle S_{3E6} \rangle\}$

$NT' = \{S_{23456}, S_{256}, S_{356}, S_{E56}, S_{234E6}, S_{2E6}, S_{3E6}\}$

For non-terminal S_{E56}

$i = 5$

$temp = X' = X = \{E,5,6\}$

$union\_left = \{2\}, union\_right = \{4\}, closure - observe = \{2,3\}$

$P \leftarrow P \cup \{\langle S_{E56} \rangle \rightarrow e\langle S_{EE6} \rangle\}$

$NT' = \{S_{23456}, S_{256}, S_{356}, S_{E56}, S_{234E6}, S_{2E6}, S_{3E6}, S_{EE6}\}$

For non-terminal S_{234E6}

$i = 2$

$temp = X' = X = \{2,3,4,E,6\}$

$union\_left = \{2\}, union\_right = \{4\}, closure - observe = \{2,3\}$

$X' = X' - j = \{2,3,E,6\}$

$i \in closure\_start$

$X' = X' - followpos(2) = \{2,3,E,6\} - \{2,3\} = \{E,6\}$

$temp1 = \{2,3\}$

$i = 2$

$P \leftarrow P \cup \{\langle S_{234E6}\rangle \rightarrow b\langle S_{2E6}\rangle\}$

$P \leftarrow P \cup \{\langle S_{234E6}\rangle \rightarrow b\langle S_{3E6}\rangle\}$

$i = 3$

$P \leftarrow P \cup \{\langle S_{234E6}\rangle \rightarrow c\langle S_{EE6}\rangle\}$

$NT' = \{S_{23456}, S_{256}, S_{356}, S_{E56}, S_{234E6}, S_{2E6}, S_{3E6}, S_{EE6}\}$

$i = 4$

$union\_left = \{2\}, union\_right = \{4\}, closure - observe = \{2,3\}$

$X' = X' - j = \{4,E,6\}$

$P \leftarrow P \cup \{\langle S_{234E6}\rangle \rightarrow d\langle S_{EE6}\rangle\}$

$NT' = \{S_{23456}, S_{256}, S_{356}, S_{E56}, S_{234E6}, S_{2E6}, S_{3E6}, S_{EE6}\}$

For non-terminal S_{2E6}

$i = 2$

$X' = X' - \{j\} = \{E,6\}$

$P \leftarrow P \cup \{\langle S_{2E6}\rangle \rightarrow b\langle S_{2E6}\rangle\}$

$P \leftarrow P \cup \{\langle S_{2E6}\rangle \rightarrow b\langle S_{3E6}\rangle\}$

$NT' = \{S_{23456}, S_{256}, S_{356}, S_{E56}, S_{234E6}, S_{2E6}, S_{3E6}, S_{EE6}\}$

For non-terminal S_{3E6}

$i = 3$

$X' = X' - \{j\} = \{E,6\}$

$$P \leftarrow P \cup \{\langle S_{3E6}\rangle \rightarrow c\langle S_{EE6}\rangle\}$$

$$NT' = \{S_{23456}, S_{256}, S_{356}, S_{E56}, S_{234E6}, S_{2E6}, S_{3E6}, S_{EE6}\}$$

On applying algorithms 6.3.4, 6.3.5 and 6.3.6

$$\langle S\rangle \rightarrow a\langle S_{23456}\rangle$$

$$\langle S_{23456}\rangle \rightarrow b\langle S_{256}\rangle|cef|def|e\langle S_{234E6}\rangle|bcef|becf$$

$$\langle S_{256}\rangle \rightarrow b\langle S_{256}\rangle|e\langle S_{2E6}\rangle|bcef|becf$$

$$\langle S_{234E6}\rangle \rightarrow b\langle S_{2E6}\rangle|bcf|cf|df$$

$$\langle S_{2E6}\rangle \rightarrow b\langle S_{2E6}\rangle|bcf$$

Example 6.4.2 Let $r = ((ab)^*\&c^*)d$

$r' = ((a_1b_2)^*\&c_3^*)\#_4d_5\#_6$

$N = \{S, S_1, S_2, ..., S_6\}$

$N' = \{\}$

$union\_left = union\_right = \{\}$

$followpos(1) = \{2\}$

$followpos(2) = \{1, E\}$

$followpos(3) = \{3, E\}$

$followpos(4) = \{5\}$

$followpos(5) = \{6\}$

$followpos(6) = \{\}$

$N' = \{S_{134}\}$

$P \leftarrow P \cup \{\langle S_{134}\rangle \rightarrow \langle S_{E34}\rangle|\langle S_{1E4}\rangle|\varepsilon\}$

$N = \{S, S_4, S_5, S_6\}$

$P \leftarrow P \cup \{\langle S\rangle \rightarrow \langle S_4\rangle\}$

$P \leftarrow P \cup \{\langle S_4\rangle \rightarrow \langle S_5\rangle\}$

$P \leftarrow P \cup \{\langle S_5\rangle \rightarrow d\langle S_6\rangle\}$

$P \leftarrow P \cup \{\langle S_6\rangle \rightarrow \varepsilon\}$

$P \leftarrow P \cup \{\langle S\rangle \rightarrow \langle S_{134}\rangle\}$

$union\_left = union\_right = \{\}$

For non-terminal S_{134}

$i = 1$

$X' = \{1,3,4\}$

$closure\_start = \{1,3\}$

$X' = X' - followpos(2) = \{3,4\}$

$P \leftarrow P \cup \{\langle S_{134}\rangle \rightarrow a\langle S_{234}\rangle\}$

$NT' = \{S_{134}, S_{234}\}$

$i = 3$

$X' = \{1,3,4\}$

$X' = X - 3 = \{1,4\}$

$P \leftarrow P \cup \{\langle S_{134}\rangle \rightarrow c\langle S_{134}\rangle\}$

$P \leftarrow P \cup \{\langle S_{134}\rangle \rightarrow c\langle S_{1E4}\rangle\}$

$NT = \{S_{134}, S_{234}, S_{1E4}\}$

For non-terminal S_{234}

$i = 2$

$X = \{2,3,4\}$

$X = X - 2 = \{3,4\}$

$P \leftarrow P \cup \{\langle S_{234}\rangle \rightarrow b\langle S_{134}\rangle\}$

$P \leftarrow P \cup \{\langle S_{234}\rangle \rightarrow b\langle S_{E34}\rangle\}$

$NT = \{S_{134}, S_{234}, S_{1E4}, S_{E34}\}$

$i = 3$

$X = \{2,3,4\}$

$X = X' - 3 = \{2,4\}$

$P \leftarrow P \cup \{\langle S_{234}\rangle \rightarrow c\langle S_{234}\rangle\}$

$P \leftarrow P \cup \{\langle S_{234}\rangle \rightarrow c\langle S_{2E4}\rangle\}$

$NT' = \{S_{134}, S_{234}, S_{1E4}, S_{E34}, S_{2E4}\}$

For non-terminal S_{1E4}

$i = 1$

$X' = \{1, E, 4\}$

$X' = X' - \{1\} = \{E, 4\}$

$P \leftarrow P \cup \{\langle S_{1E4}\rangle \rightarrow a\langle S_{2E4}\rangle\}$

$NT' = \{S_{134}, S_{234}, S_{1E4}, S_{E34}, S_{2E4}\}$

For non-terminal S_{E34}

$i = 3$

$X' = \{E, 3, 4\}$

$X' = X' - \{3\} = \{E, 4\}$

$P \leftarrow P \cup \{\langle S_{E34}\rangle \rightarrow c\langle S_{E34}\rangle\}$

$P \leftarrow P \cup \{\langle S_{E34}\rangle \rightarrow c\langle S_{EE4}\rangle\}$

$NT' = \{S_{134}, S_{234}, S_{1E4}, S_{E34}, S_{2E4}, S_{EE4}\}$

For non-terminal S_{2E4}

$i = 2$

$X' = \{2, E, 4\}$

$X' = X' - \{2\} = \{E, 4\}$

$P \leftarrow P \cup \{\langle S_{2E4}\rangle \rightarrow b\langle S_{1E4}\rangle\}$

$P \leftarrow P \cup \{\langle S_{2E4}\rangle \rightarrow b\langle S_{EE4}\rangle\}$

$NT' = \{S_{134}, S_{234}, S_{1E4}, S_{E34}, S_{2E4}, S_{EE4}\}$

On applying algorithm 6.3.4, 6.3.5 and 6.3.6

$\langle S\rangle \rightarrow d|\langle S_{134}\rangle$

$\langle S_{134}\rangle \rightarrow a\langle S_{234}\rangle|c\langle S_{134}\rangle|c\langle S_{1E4}\rangle|\langle S_{E34}\rangle|\langle S_{1E4}\rangle|\varepsilon$

$\langle S_{234}\rangle \rightarrow b\langle S_{134}\rangle|b\langle S_{E34}\rangle|c\langle S_{234}\rangle|c\langle S_{2E4}\rangle$

$\langle S_{1E4}\rangle \rightarrow a\langle S_{2E4}\rangle$

$\langle S_{E34}\rangle \rightarrow c\langle S_{E34}\rangle|cd$

$\langle S_{2E4}\rangle \rightarrow b\langle S_{1E4}\rangle|bd$

6.5. CONCLUSION

In this chapter, an algorithm is proposed for the metamorphosis of *PRE* to regular grammar. It is a first attempt of its kind in this direction. Although, computational algorithm has been designed by other authors [3, 20] for the metamorphosis of regular expression to *DFA* using the *followpos* of symbols. The novelty of the algorithm is the use of *followpos* for the metamorphosis of *PRE* to regular grammar. The numerical implementation of the algorithm is outlined, and specific numerical examples are given to illustrate the proposed algorithm. Further, the grammar procured from the proposed algorithm can be easily converted into a minimized *DFA*. This algorithm can also be beneficial for studying and analyzing various properties of the *PRE*.

Chapter 7

AN IMPROVED ALGORITHM FOR THE METAMORPHOSIS OF SEMI-EXTENDED REGULAR EXPRESSIONS TO DETERMINISTIC FINITE AUTOMATA

7.1 INTRODUCTION

Semi-extended regular expressions (*SEREs*) consist of intersection, union, concatenation and Kleene closure operators. Comprisal of intersection operators in *REs* makes it comfy to use. In many applications, the words depicted by the patterns are long and cannot be easily described by the *REs*. These patterns can easily be depicted by using *SERE*.

Yamamoto [31] proposed an algorithm for the translation of *SEREs* into non-deterministic finite automata(*NFAs*). Yamamoto [30] proposed an algorithm for the membership of *SEREs* using its metamorphosis to partially input-synchronized alternating automata.

Kupferman and Zuhovitzky [54] proposed the improved algorithm for the membership of *SERE* using the concept of alternating automata with synchronized universality and negation. It is arduous for metamorphosis a *SERE* into alternating automaton whenever sub-expression $(r_1 \cap r_2)r_3$ occurs in the *SERE*. Problem occurs because sub-expression $(r_1 \cap r_2)r_3$ is not equal to $(r_1r_3 \cap r_2r_3)$whereas $(r_1 \cup r_2)r_3$ is equal to $(r_1r_3 \cup r_2r_3)$.

In recent years, lot of research is targeted towards the succinctness of *REs* with additional operators like intersection, shuffle [24-25, 74-76]. They reasserted that double exponential size is required for construction of *RE* from *RE* with intersection operators. There exists a number of membership algorithms, for checking whether w can be generated from the *SERE* or not. Useful membership algorithms for *SERE* has been provided by Yamamoto [30] and extensively improved by Kuperferman and

Table 7.1.1: Comparison between direct conversion, Thompson construction and Glushkov automata

Criterion	**Thompson's Construction [43]**	**Glushkov Automata [72]**	**Direct Conversion [20]**
States	At most $2n$ states	Exactly $m+1$ states	At-most $m+1$ states
Output	ε-*NFA*	*NFA*	*DFA*

Zuhovitzky [54], Peterson [29], Rosu [22]. For a given *RE* r of length n and m be the number of symbols occur in r. Table 7.1.1 shows the comparison between three approaches for the conversion of *RE* to *DFA*.

Table 7.1.1 shows that the direct metamorphosis method gives less number of states than the other two approaches for the conversion of *RE* to finite automaton. It produces a *DFA* while the other two approaches generate *NFA*. Yamamoto algorithm for the metamorphosis of *SERE* to *NFA* is based on the Glushkov automata [72]. There is a possibility of generating a compact finite automaton using the direct conversion method [20] for converting a *SERE* to finite automaton. Motivation for this research stems from the importance of *SEREs* in the field of *XML* theory and programming language [19, 46, 64, 73]. Given two deterministic finite automata (*DFAs*) N_1 and N_2 with m and n states respectively. *DFA* corresponding to $N_1 \cap N_2$ requires up to mn states.

In this chapter an algorithm is designed for the metamorphosis of *SEREs* to *DFA*. The proposed algorithm is based on the follow-positions of symbols present in the *SERE*. Comparison demonstrates that the *DFA* generated using the proposed algorithm is smaller than the earlier existing approaches in the literature. Finally, to expound this metamorphosis, we endow some numerical examples.

7.2 PRELIMINARIES AND NOTATION

DEFINITION 7.2.1. Given two automata $M_1 = (\Sigma, Q_1, q_{01}, F_1, \delta_1)$ and $M_2 = (\Sigma, Q_2, q_{02}, F_2, \delta_2)$. Intersection of M_1 and M_2 can be represented by an automaton $M = (\Sigma, Q, q_0, F, \delta)$ defined as follows:

1. $Q \leftarrow Q_1 \times Q_2$
2. $q_0 \leftarrow (q_{01}, q_{02})$
3. $F \leftarrow F_1 \times F_2$
4. $\delta((q_1, q_2), a) \leftarrow (\delta(q_1, a), \delta(q_2, a))$

In this chapter, a metamorphosis of *SEREs* to *DFAs* algorithm (*MSEREDFA*) is evolved using the *followpos* of symbols. For a sketch of how the proposed algorithm is executing, we bestow numerical examples. The detail of *MSEREDFA* algorithm is introduced as follows:

1. **Augmented *SEREs*:** Add # at the cessation of *SERE r*. If $(r_i \cap r_j) \in r$, add # after r_i and r_j . Assign position to the symbols of the augmented *SERE*.
2. **Calculation of *followpos*:** Using *Calc_follow*, *followpos* of position are determined. In *Calc_follow*, definitions of *firstpos*, *lastpos*, *followpos*, and *nullable* at the union, concatenation and Kleene closure with the minor alterations occur due to intersection operators. $fpos_{Inter}$ depicts a set containing the first positions of all sub-expression $(r_i \cap r_j) \in r$.

   At intersection operator:

$$firstpos(\cap) = firstpos(r_i) \cup firstpos(r_j)$$

$$lastpos(\cap) = lastpos(r_i) \cup lastpos(r_j)$$

$$nullable(\cap) = nullable(r_i) \wedge nullable(r_j)$$

$$fpos_{Inter} = fpos_{Inter} \cup firstpos(r_i\# \cap r_j\#)$$

3. **Construction of *DFA*:** *firstpos*(*r*') render the starting state of the *DFA*. New states are framed on reading of symbols using the concept of *followpos*. Each state is partitioned into two sets of positions. First set consists of the positions which are not involved in the intersection and the other one consists of the positions involved in the intersection operation. *DFA* is constructed using the procedure *Intersection_DFA* for the positions of a state involved in shuffling.

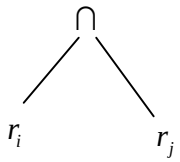
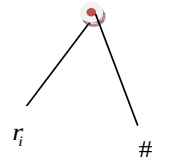
Using the analysis described above an algorithm *MSEREDFA* is designed for the metamorphosis of *SEREs* to *DFAs*.

Notation used in the algorithm: Throughout this chapter, *node* represents the current symbol. It can be one of the symbols from Σ, an operator or # symbol. L and R depict the two children of a node in the syntax tree. Single child of a node is depicted by L.

Procedure *Calc_follow* will receive an augmented *SERE* as input and determine *firstpos*, *lastpos* and *nullable* at each node. This procedure will also determine the *followpos* of the positions. Additional rules are given in Table 7.2.1 that occurs due to the intersection operator.

Intersection_DFA will produce the states of the *DFA* occurs due to the intersection operators. Partition state q into sets C and D such that C contains the positions of r_i and D contains the positions of r_j for the sub-expression $(r_i \cap r_j) \in r$. For each symbol a, *followpos* of C and D are determined. If both are not null, then a new state is generated by taking the union of *followpos* of C and D at symbol a. This process is repeated until we have an unmarked state occurs due to the current intersection operator. The final state (say q_I) of the intersection operator contains the positions of both #'s occur after r_i and r_j. Before returning to the calling function, *MSEREDFA* forge q_I as unmarked state and q_I is substituted by the *followpos* of # of the sub-expression $(r_i\# \cap r_j\#)$.

Table 7.2.1: Additional Rules occurring due to the Intersection operator

Syntax tree	Current node	Rules
	Intersection operator	$firstpos(node) \leftarrow firstpos(L) \cup firstpos(R)$ $lastpos(node) \leftarrow lastpos(L) \cup lastpos(R)$ $nullable(node) \leftarrow nullable(L) \wedge nullable(R)$ $fpos_{Inter} = fpos_{Inter} \cup firstpos(node)$
	Concatenation operator	$nullable(node) = (nullable(L) \wedge nullable(R))$ If($nullable(node)$=$false$) $nullable(node) = (nullable(L) \wedge (R = \#))$

7.3 Algorithm for Metamorphosis of SEREs to DFAs

In this section the algorithm for the metamorphosis for the conversion of *SEREs* to *DFAs* is given.

Algorithm 7.3.1: *MSEREDFA*(r)

Input: *SERE* r
Output: *DFA* M *such that* $L(M) = L(r)$.

1 Augmented *SERE*: $r' \leftarrow r\#$
2 **for** $(\forall(r_i \cap r_j) \in r')$ **do**
3 $\quad r_i \leftarrow r_i\#$
4 $\quad r_j \leftarrow r_j\#$
5 **end**
6 Assign position to the symbols of r'.
7 $Q \leftarrow fpos_{Inter} \leftarrow \emptyset$
8 $Calc\_follow(r')$
9 $q_0 \leftarrow firstpos(r')$

Algorithm 7.3.1: *MSEREDFA*(r) **Continued...**

```
10 Q ← Q ∪ q_0
11 for (∀q ∈ Q) do
12 |  Mark q
13 |  q' ← ∅
14 |  q'' ← q
15 |  for (∀i ∈ q) do
16 |  |  if (i ∈ fpos_Inter) then
17 |  |  |  q' ← q' ∪ i
18 |  |  |  q'' ← q − i
19 |  |  end
20 |  end
21 |  if (q' ≠ ∅) then
22 |  |  Intersection_DFA(q, Q)
23 |  end
24 |  if (q'' ≠ ∅) then
25 |  |  for (∀a ∈ Σ) do
26 |  |  |  Qstate ← ∅
27 |  |  |  for (∀i ∈ q'') do
28 |  |  |  |  if (a_i = a) then
29 |  |  |  |  |  Qstate ← Qstate ∪ followpos(i)
30 |  |  |  |  end
31 |  |  |  end
32 |  |  |  if Qstate ≠ ∅ then
33 |  |  |  |  Q ← Q ∪ Qstate
34 |  |  |  |  trans[q, a] ← Qstate
35 |  |  |  end
36 |  |  end
37 |  end
38 end
```

Algorithm 7.3.2: *Calc_follow*(r')

Input: An augmented *SERE* r' .
Output: $fpos_{Inter}$, *firstpos*, *lastpos and followpos*.

1 Draw a syntax tree for r' such that the operators appear as the internal nodes and symbols from Σ and #'s appear as the leaf node.

2 Traverse the syntax tree in post-order and perform step 3 to 7. **Continued...**

Algorithm 7.3.2: *Calc_follow*(r') **Continued...**

```
3  if ((node = a) ∧ (a ∈ Σ)) then
4  |  firstpos(node) ← position(node)
5  |  lastpos(node) ← position(node)
6  |  nullable(node) ← false
7  end
8  if (node = |) then
9  |  firstpos(node) ← firstpos(L) ∪ firstpos(R)
10 |  lastpos(node) ← lastpos(L) ∪ lastpos(R)
11 |  nullable(node) ← nullable(L) ∨ nullable(R)
12 end
13 if (node = *) then
14 |  firstpos(node) ← firstpos(L)
15 |  lastpos(node) ← lastpos(L)
16 |  nullable(node) ← true
17 |  for (∀j ∈ lastpos(node)) do
18 |  |  followpos(j) ← followpos(j) ∪ firstpos(node)
19 |  end
20 end
21 if (node = ∩) then
22 |  firstpos(node) ← firstpos(L) ∪ firstpos(R)
23 |  lastpos(node) ← lastpos(L) ∪ lastpos(R)
24 |  nullable(node) ← nullable(L) ∧ nullable(R)
25 |  fpos_Inter ← fpos_Inter ∪ firstpos(node)
26 end
27 if (n = .) then
28 |  firstpos(node) ← firstpos(L)
29 |  lastpos(node) ← lastpos(R)
30 |  nullable(node) ← (nullable(L) ∧ nullable(R)) ∨ (nullable(L) ∧ R = #)
31 |  if (nullable(R) = true) then
32 |  |  lastpos(node) ← lastpos(node) ∪ lastpos(L)
33 |  end
34 |  if (nullable(L) = true) then
35 |  |  firstpos(node) ← firstpos(node) ∪ firstpos(R)
36 |  end
37 |  for (∀j ∈ lastpos(L)) do
38 |  |  followpos(j) ← followpos(j) ∪ firstpos(R)
39 |  end
40 end
```

Algorithm 7.3.3: *Intersection_DFA*(q, Q)

Input: state q
Output: Addition of states in Q

```
1  Q_Inter ← q
2  for (∀q ∈ Q_inter) do
3      C ← D ← ∅
4      Mark q
5      for (∀i ∈ q) do
6          if ((i ∈ pos(r_i#)) ∧ ((r_i# ∩ r_j#) ∈ r')) then
7              C ← C ∪ i
8          end
9          if ((i ∈ pos(r_j#)) ∧ ((r_i# ∩ r_j#) ∈ r')) then
10             D ← D ∪ i
11         end
12     end
13     for (∀a ∈ Σ) do
14         E ← F ← ∅
15         for (∀i ∈ C) do
16             if (a_i = a) then
17                 E ← E ∪ followpos(i)
18             end
19         end
20         for (∀i ∈ D) do
21             if (a_i = a) then
22                 F ← F ∪ followpos(i)
23             end
24         end
25         if ((E ≠ ∅) ∧ (F ≠ ∅) ) then
26             if ({E/F} ∉ Q_inter) then
27                 Q_inter = Q_inter ∪ {E/F}
28             end
29             trans[q, a] ← {E/F}
30         end
31     end
32 end
33 Choose q such that ((q ∈ Q_inter) ∧ (pos(#(r_i#))) ∧ (pos(#(r_j#))))
34 q ← followpos(#(r_1#)) ∪ followpos(#(r_2#))
35 Unmark q
36 Q ← Q ∪ Q_inter
```

Limitation of the Algorithm: In the following two cases, the proposed algorithm converts a *SERE* into *ε-free NFA* instead of *DFA*.

1. If $(r_i \cap r_j)r_k \in r'$and the following conditions exist:

(a) $nullable(r_i \cap r_j) = true$

(b) $(symbols_{lastpos(r_i \cap r_j)}) \cap (symbols_{firstpos(r_k)}) \neq \emptyset$

2. If $r_k(r_i \cap r_j) \in r'$ and the following conditions exist:

(a) $nullable(r_k) = true$

(b) $(symbols_{lastpos(r_k)}) \cap (symbols_{firstpos(r_i \cap r_j)}) \neq \emptyset$

EXAMPLE 7.3.1. Given $r = a(((a|b)^*a(a|b)^*a(a|b)^*) \cap (a^*ba^*ba^*))b$

On enforcing the proposed algorithm on r, we obtain the *DFA* as shown in Figure 7.3.1.

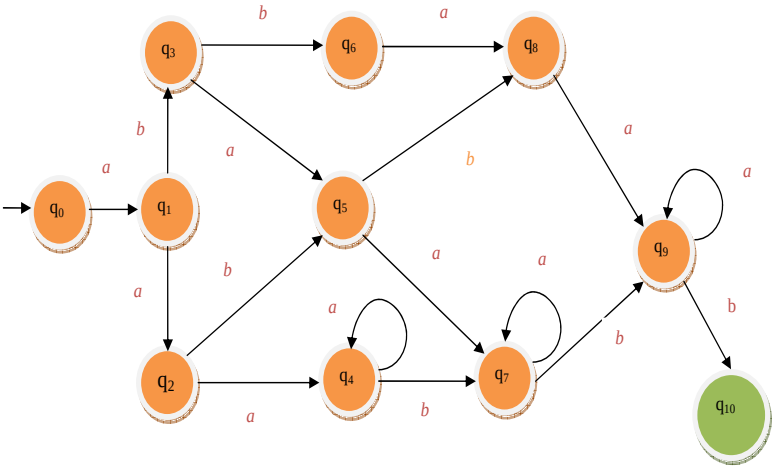


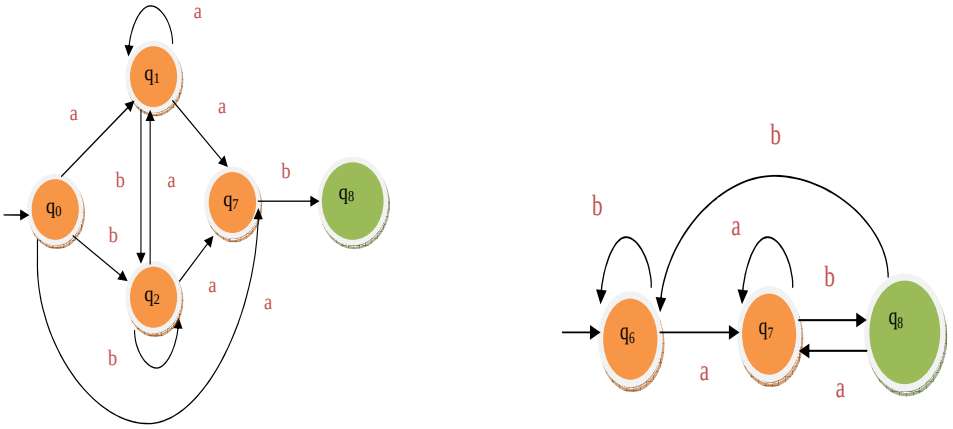
Figure 7.3.1: *DFA* for $r = a(((a|b)^*a(a|b)^*a(a|b)^*) \cap (a^*ba^*ba^*))b$

7.4 Results and Discussion

Using Yamamoto algorithm [31], *SERE* can be converted into a ε-*NFA*. Pattern matching will be delayed if the *NFA* consists of ε-transitions. Yamamoto algorithm is based on the Glushkov automata [55]. The number of states in the Glushkov automata

is equal to $|r|+1$. The proposed approach is based on the direct conversion of *REs* to *DFAs*. In direct conversion method [20], numbers of states are reduced due to the occurrence of each Kleene closure and union operators. For conversion of *RE* to *finite automaton*, Glushkov automata generate more states than the direct conversion method. *SERE* can be converted into *DFA* using the proposed approach.

EXAMPLE 7.4.1. Consider $r = (a|b)^*ab$. Figure 7.4.1 illustrate the finite automata obtained using the Glushkov automata and direct conversion method. The number of states in a finite automaton obtained using the direct conversion method is less than the number of states obtained using the Glushkov automata.



(a) Glushkov automata **(b)** DFA using the direct conversion method

Figure 7.4.1: *RE* $r = (a|b)^*ab$

Following example, illustrate the result obtained by the proposed algorithm for converting a *SERE* to *DFA*. At the same time, the result of Yamamoto algorithm is presented for comparing the size of the automaton.

EXAMPLE 7.4.2. Given $r = 1(((00)^*) \cap ((000)^*))1$. Using the proposed algorithm, the *DFA* generated is shown in Figure 7.4.2.

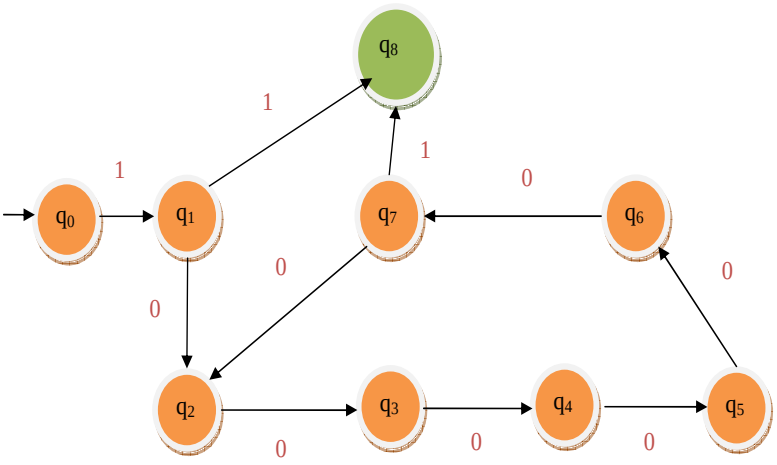


Figure 7.4.2: *DFA* for *SERE* $r = 1(((00)^*) \cap ((000)^*))1$

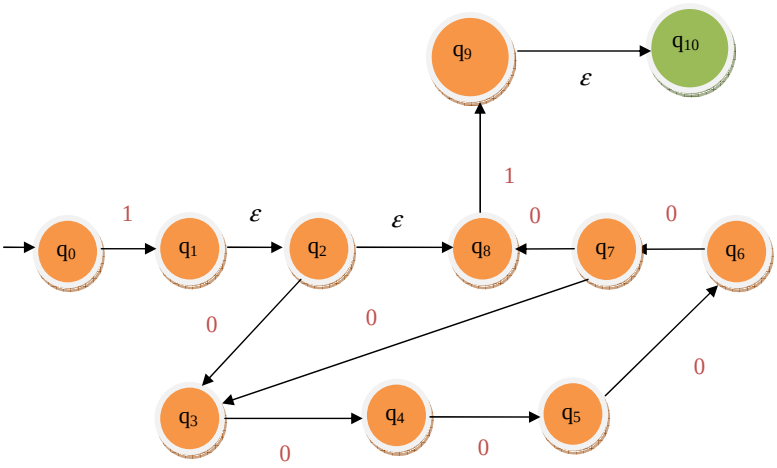


Figure 7.4.3: *NFA* for $r = 1(((00)^*) \cap ((000)^*))1$ using Yamamoto Methodology

Figure 7.4.3 shows the ε-*NFA* obtained using the Yamamoto Methodology.

Table 7.4.1: Comparison between Yamamoto algorithm and proposed algorithm

Criterion	**Yamamoto algorithm**	**Proposed Algorithm**
Output	ε-*NFA*	*DFA*
Approach	Based on Glushkov Automata	Based on direct conversion of *REs* to *DFAs*.
Number of states	More states	Less states

Table 7.4.1 shows the comparison between the two approaches.

Yamamoto algorithm is founded on the modular decomposition of r. The number of states using the Yamamoto algorithm [31] is at most N_r where N_r is equal to $\sum_{h\in H}\prod_{R\in h}(N+1)$ where

1. R is the set of modules obtained from the *SERE* r.
2. H is the accumulation of all independent subset of R.
3. N is the total number of symbols and intersection operators occur in R.

Now, we claim that the number of states of the *DFA* generated using the proposed approach is less than the number of states of the *NFA* generated using Yamamoto algorithm. In the proposed methodology, N be the total number of symbols occurs in R. Following two cases can occur:

1. For each unyoked module, the number of states obtained using the proposed approach is less than the Yamamoto algorithm. In unyoked module, each sub-expression is a *RE*. Table 7.1.1 show, that the number of states of the *DFA* generated by direct conversion is less than *NFA* generated using the Glushkov automata.
2. Let $H_i \in H$ be a subset whose sub-expressions are involved in the intersection operators and $H_i = \{r_1, r_2, ...\}$. For each $r_i \in H_i$, the number of

states generated using the direct conversion method is less than the Glushkov automata (as shown in Table 7.1.1). Each intersection operation performed between the *NFA* obtained for r_i causes the addition of two new states and a number of transitions in the Yamamoto approach, which is not added in the proposed approach. Hence we claim that the number of states of the *DFA* obtained using the proposed approach is less than the number of states of the *NFA* generated using Yamamoto algorithm [31].

7.5 CONCLUSION

In this chapter, an algorithm for direct metamorphosis of *SEREs* to *DFAs* has been proposed. This algorithm is based on the direct conversion of *REs* to *DFAs*. Major benefit of the proposed algorithm is the generation of a svelte *DFA*. Another major benefit of the proposed algorithm is that it will not face any problem if *SERE* contains sub-expression $(r_1 \cap r_2)r_3$. The numerical implementation of the proposed algorithm is expounded by numerical examples in Sections 7.3 and 7.4. The major contribution can be highlighted as this being on novel concept, one of its own kind, never explored before. Further, using this approach the number of states of the *DFA* generated using the proposed algorithm is less than the ε-*NFA* generated using the existing approaches till date.

Chapter 8

CONCLUSION AND FUTURE SCOPE

8.1 Summary of the Main Contribution

In this dissertation, we proposed three algorithms in the field of parallel regular expressions and semi-extended regular expressions. The first proposed algorithm (*MPRENFA*) converts parallel regular expressions into svelte non-deterministic finite automata and proves that if $\forall(r_i \& r_j) \in r \wedge ((symbol(r_i) \cap symbol(r_j)) \neq \emptyset)$, then it is implausible to generate a deterministic finite automaton directly from parallel regular expression. The number of states of the *NFA* generated using the proposed approach is 2^{m+1}, which is a significant improvement over the Estrade *et al.* [6] approaches. The obtained *NFA* can be converted into *DFA* using subset construction.

Our second proposed algorithm (*MSEREDFA*) converts semi-extended regular expressions to deterministic finite automata. The proposed algorithm is founded on the direct conversion of *REs* to *DFAs* [20]. Further, It has been proved that the number of states obtained in deterministic finite automaton (*DFA*) is less than the size of non-deterministic finite automaton (*NFA*) obtained using Yamamoto algorithm [30], which is a significant contribution in the field of automata theory.

The third algorithm (*PRE_RLG*) is proposed for the metamorphosis of *PRE* to regular grammar. To the best of the author's knowledge, no such metamorphosis has ever been proposed. The grammar procured from the proposed algorithm can be easily converted into a minimized *DFA*. The metamorphosis of parallel regular expressions to regular grammar gives a prescript of reduction in the field of concurrency. The contribution will open new avenues for the research in this field.

8.2 PRESPECTIVES

Following are a number of promising and stimulating directions for future research building on the proposed work and gleaning of knowledge presented in this dissertation.

8.2.1 CONVERSION OF EXTENDED REGULAR EXPRESSIONS TO *NFA*

One main goal in this thesis is the conversion of parallel regular expressions to non-deterministic finite automata. Another goal in this thesis is the metamorphosis of semi-extended regular expressions to deterministic finite automata. This will leads to another interesting problem for the conversion of extended regular expressions (regular expressions having shuffle and intersection operators) to non-deterministic finite automata.

8.2.2 FURTHER OPTIMIZATIONS IN TERM OF THE SIZE OF *NFA*

Although, the size of the non-deterministic finite automata obtained using the proposed approach is smaller than the existing approaches in the literature, we cannot say that it is the minimal one, and there is still a scope of optimization in terms of the number of states.

8.2.3 CONVERSION OF *PRE* TO *RE*

There is a need of more precise and credible choice for the conversion of parallel regular expressions to regular expressions. However, conversion of parallel regular expressions to partial serialization can be useful in the field of modeling of concurrent systems.

8.2.4 AUTOMATION OF THE DESIGNED ALGORITHMS

Parallel regular expressions are useful in different fields of Computer Sciences. There is a scope of the automation of the newly designed algorithms.

8.2.5 Comparison Between the Novel Algorithms

In chapter *6*, we have designed an algorithm for the conversion of parallel regular expressions to right linear grammar. There is a scope for the conversion of right linear grammar to minimized deterministic finite automaton, which will lead to another interesting direction for the conversion of parallel regular expressions to deterministic finite automata. This work can be extended for the comparison between the two proposed approaches ($PRE \rightarrow NFA \rightarrow DFA$ and $PRE \rightarrow$ *right linear grammar* $\rightarrow DFA$).

8.2.6 Time Complexity of Novel Algorithms

All three designed algorithms work better in terms of the number of states than the existing approaches in the literature. However, there is a need to work on another parameter (time complexity) for the newly developed algorithms.

References

[1] A. Bruggemann-Klein, Regular expressions into finite automata, *Theoretical Computer Science,* 120(2), (1993), 197-213.

[2] A. Meduna, Elements of compiler designs, *Auerbach Publications,* (2007).

[3] A. V. Aho, R. Sethi and J. D. Ullman, Compilers: principles, techniques and tools, *Pearson Education*, (2005).

[4] A virus Scanning System, Clam Anti Virus: open source anti-virus toolkit, *http://www.clamav.net/lang/en*.

[5] B. D. Estrade, An investigation of equivalent serialized forms of parallel finite automata, *Master's Thesis, The University of Southern Mississippi*, (2005).

[6] B. D. Estrade, A. L. Perkins and J. M. Harris, Explicitly parallel regular expressions, *In proc. of the first International multi-symposiums on computer and computational sciences (IMSCCS'06), IEEE*, (2006), 402-409.

[7] B. Pardeep and C. S. R. Murthy, A constant time string shuffle algorithm on reconfigurable meshes, *International Journal of Computer Mathematics,* 68(3-4), (1998), 251-260.

[8] C. Campeanu, K. salomaa and S. Vagvolgyi, Shuffle quotient and decompositions, *In Proc. of* 5^{th} *international conference in developments in Language Theory, LNCS, Springer, Wien, Austria,* 2295, (2002), 223-226.

[9] C. M. Sperberg-McQueen and H. Thompson, XML schema, *http://www.w3.org/TR/xmlschema11-2/*, 2005.

[10] C. Schmidt, N. Sridharan and J. Goodson, The Plan recognition problem: An Intersection of psychology and artificial intelligence,*Artificial Intelligence*, 11(1-2), (1978).

[11] D. Giammarresi, J. L. Ponty, D. Wood, and D. Ziadi, A characterization of Thompson digraphs, *Discrete Applied Mathematics*, 134(1), (2004), 317-337.

[12] 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, (2012), 313-324.

[13] D. Hovland, Feasible Algorithms for Semantics-Employing Automata and Inference Systems, *PhD Thesis*, The University of Bergen, Norway, (2010).

[14] D. Micciancio, The validity problem for extended regular expressions, *M. S. Thesis*, Massachusetts Institute of Technology, (1996).

[15] D. W. Mount, Bioinformatics: Sequence and Genome analysis, *Cold Spring Harbor Laboratory Press*, 2^{nd} Edition, 2004.

[16] D. Ziadi, and J. Champarnaud, An optimal parallel algorithm to convert a regular expression into its Glushkov automaton, *Theoretical computer science*, 215(1), (1999), 69-87.

[17] E. Lughofer and S. Kindermann, Improving the Robustness of Data-Driven Fuzzy-Systems with Regularization, *In the Proc. of the IEEE World Congress on Computational Intelligence (WCCI), Hongkong*, (2008), 703-709.

[18] F. Biegler, M. Daley, M. Holzer and I. McQuillan, On the uniqueness of shuffle on words and finite languages, *Theoretical Computer Science,* 410(38-40), 2009, 3711-3724.

[19] F. Neven, Automata, Logic and XML, *In Proc. of the 16th International Workshop on Computer Science Logic (CSL 02)*, Lecture Notes in Computer Science Springer-verlag Berlin, 2471, (2002), 2-26.

[20] G. Berry and R. Sethi, From Regular expressions to deterministic Automata, *Theoretical Computer Science*, 48, (1986), 117-126.

[21] G. Castiglione, A. Restivo, and M. Sciortino, Nondeterministic Moore Automata and Brzozowski's minimization Algorithm, *Theoretical Computer Science*, 450, (2012), 81-91.

[22] G. Rosu, An effective algorithm for the membership problem for extended regular expressions, *In Proc. of FOSSACS*, *Lecture Notes in Computer Science Springer-verlag, Berlin*, 4423, (2007), 332-345.

[23] H. Bjorklund, and M. Bojanczyk, Shuffle expressions and words with nested data, *In Proc. of Mathematical Foundations of Computer Science, LNCS, Springer Berlin,* 4708, (2007), 750-761.

[24] H. Gruber and M. Holzer, Finite automata, digraph connectivity and regular expression size, *In Proc. of the 35th International Colloquium on Automata, Languages and Programming, LNCS springer-verlag Berlin Heidelberg*, 2, 2008, 39-50.

[25] H. Gruber, and M. Holzer, Tight bounds on the descriptional complexity of regular expressions, *Springer Berlin Heidelberg, Developments in Language Theory*, (2009), 276-287.

[26] *http://www.math.hawaii.edu/ pavel/bernoulli.pdf*

[27] *http://www.math.upenn.edu/ wilf/PIMS/PIMSLectures.pdf*

[28] *http://search.cpan.org/ estrabd/FLAT-0.9.1/lib/FLAT.pm*

[29] H. Petersen, The membership problem for regular expressions with intersection is complete in LOGCFL, *In Proc. of the 19th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2002), LNCS Springer-verlag Berlin*, 2285, (2002), 513-522.

[30] H. Yamamoto, An automata-based recognition algorithm for semi-extended regular expressions, *In Proc. of MFCS'00, LNCS Springer-verlag Berlin*, 1893, (2000), 699-708.

[31] H. Yamamoto, *A new Translation from semi-extended Regular Expressions into NFA and Its Application to an Approximate Matching Problem,* In Proc. of ISAAC'03, LNCS Springer-verlag Berlin, 2906, (2003), 158-167.

[32] Intrusion detection systems, Firekeeper: Detect and block malicious sites http://firekeeper.mozdev.org/

[33] J. A. Brzozowski, Derivatives of regular expressions, *Journals of the Association for Computing Machinery,* 11(4), (1964), 481-494.

[34] J. Berstel, L. Boasson, O. Carton, J. Pin and A. Restivo, The expressive Power of the shuffle product, *Information and Computation*, 208(11), (2010), 1258-1272.

[35] J. Clark and M. Murata, RELAX NG specifications, *Available: http://relaxng.org/spec-20011203.html,* 2001.

[36] J. C. M. Baeten and W. P. Weijland, Process algebra, *Cambridge tracts in theoretical computer science, Cambridge University press, Cambridge,* 18, (1990), 613-618.

[37] J. E. Hopcroft, R. Motwani and J. D. Ullman, Introduction to automata theory, Languages and computation, *Pearson Education,* (2005).

[38] J. Hogberg and L. Kaati, Weighted unranked tree automata as a framework for plan recognition, *In Proc. of the International Conference on Information Fusion' 10,* (2010).

[39] J. M. Champarnaud, D.Ziadi, Computing the equation automaton of a regular expression in O(s^2) space and time, *In proc. of the 12th Combinatorial Pattern Matching, LNCS Springer-Verlag, Berlin*, 2089, (2001), 157-168.

[40] K. Ellul, B. Krawetz, J. Shallit, and M. W. Wang, Regular expressions: New results and open problems, *Journal of Automata Languages and Combinatorics*, 10(4), (2005), 407-437.

[41] 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,* (1983), 374-381.

[42] K.R. Abrahamson, Succint Representation of Regular Sets using Gotos and Boolean Variables, *J. Computer and System sciences,* 34(1), (1987), 129-148.

[43] K. Thompson, Regular expression search algorithm, *Communications of the ACM,* 11(6), (1968), 419-422.

[44] L. Ilie and S. Yu., Follow automata, *Information and Computation*, 186(1), (2003), 146-162.

[45] L. Ilie and S. Yu, Reducing NFAs by invariant equivalences, *Theoretical computer science*, 306(1), (2003), 373-390.

[46] L. Wall, T. Christiansen and J. Orwant, Programming Perl, *O'Reilly*, 3, (2000).

[47] M. Bala and L. Awasthi, Proficient D-HEED Protocol for Maximizing the Lifetime of WSN and Comparative Performance Investigations with Various Deployment Strategies, *International Journal of Advanced Science and Technology*, 45, (2012), 107-123.

[48] M. Berglund, H. Bjorklund, and J. Hogberg, Recognizing Shuffled Languages, *In Proc. of LATA 2011, LNCS springer-verlag Berlin,* 6638, (2011), 142-154.

[49] M. Domaratzki and K. Salomaa, Restricted sets of trajectories and Decidability of shuffle decompositions, *International Journal of Foundations of Computer Science (IJFCS),* 16(5), (2005), 897-912.

[50] M. H. Ter Beek and J. Kleijn, Infinite unfair shuffles and associativity, *Theoretical Computer Science,* 380(3), (2007), 401-410.

[51] M. Ito, Shuffle Decomposition of Regular Languages, *Journal of Universal Computer Science,* 8(2), (2002), 257-259.

[52] M. Mishna and M. Zabrocki, Analytic Aspects of the shuffle product, *Symopsium of Theoretical Aspects of Computer Science Bordeaux,* (2008), 561-572.

[53] M. Nivat , Behaviors of synchronized systems of processes, *L.I.T.P. Report no. 81-64 University de Paris 7,* (1981).

[54] O. Kupferman and S. Zuhovitzky, An improved algorithm for the membership problem for extended regular expressions, *In Proc. of MFSC02, LNCS Springer-verlag Berlin*, 2420, (2002), 446-458.

[55] P. Caron and D. Ziadi, Characterization of Glushkov Automata, *Theoretical Computer Science*, 233(1-2), (2000), 75-90.

[56] P. D. Stotts and W. Pugh, Parallel finite automata for modeling concurrent software systems, *Journals of Systems and Software*, 27(1), (1994), 27-43.

[57] P. Garcia, D. Lopez, J. Ruiz and G. I. Alvarez, From regular expressions to smaller NFAs, *Theoretical Computer Science*, 412, (2011), 5802-5807.

[58] R. K. Aggarwal and M. Dave, Application of genetically optimized neural networks for hindi speech recognition system, *In Proc. of World Congress on Information and Communication Technologies (WICT), IEEE*, (2011), 512-517.

[59] R. McNaughton, H. Yamada, Regular expressions and state graphs for automata, *IEEE Trans. on Electronic Computers*, 9(1), (1960), 39-47.

[60] S. Broda, A. Machiavelo, N. Moreira, and R. Reis, The average transition complexity of Glushkov and partial derivative automata, *International Journal of Foundations of Computer Science*, 23(5), (2012), 969-984.

[61] S. Carberry, Techniques for plan recognition,*User Modeling and User Adapted interaction*, 11(1-2), (2001), 31-48.

[62] S. C. Kleene, Representation of events in nerve nets and finite automata, *In Shannon and McCarthy editors, Automata Studies, Princeton University Press*, (1956), 3-41.

[63] S. Kushnarev and A. Narayan, Approximating the Weil-Petersson Metric Geodesics on the Universal Teichmuller Space by Singular Solutions, *arXiv:1208.2022v2* [math.CV] 19 Oct 2012, (2012).

[64] S. Mukherjee, Satakshi and R.C. Mittal, Model order reduction using response-matching technique, *Journal of Franklin Institute*, 342(5), (2005), 503-519.

[65] S. Sippu, E. Soisalon-soininen, Parsing theory: I, languages and parsing, EATCS monographs on theoretical computer science, *Springer-Verlag, New York*, 15, (1988).

[66] S. V. K. Gaddam and M. Lal, Efficient Cancellable Biometric Key Generation Scheme for Cryptography, *International Journal of Network Security*, 11(2), (2010), 61-69.

[67] T. Schwentick, Automata for XML-a survey, *Journal of Computer and System Sciences*, 73(3), (2007), 289-315.

[68] V. Antimirov, Partial derivatives of regular expressions and finite automaton constructions, *Theoretical Computer Science*, 155, (1996), 291-319.

[69] 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).

[70] V. K. Garg and M. Ragunath, Concurrent regular expressions and their relationship to petri nets,*Theoretical Computer Science*, 96(2), (1989), 285-304.

[71] V. M. Antimirov, and P. D. Mosses, Rewriting extended regular expressions, *Theoretical Computer Science*, 143(1), (1995), 51-72.

[72] V. M. Glushkov, The abstract theory of automata, *Russian Mathematical surveys*, 16(5), (1961), 1-53.

[73] V. Vianu, Logic as a query language: from Frege to XML, *In Proc. of STACS 2003, Lecture Notes in Computer Science Springer*, 2607, (2003), 1-12.

[74] W. Gelade, Succinctness of regular expressions with interleaving,intersection and counting, *Theoretical Computer Science*, 411(31-33), (2010), 2987-2998.

[75] W. Gelade and F. Neven, Succinctness of the complement and intersection of regular expressions, *ACM Transactions on Computational Logic (TOCL)*, 13(1), 2012, 121-139.

[76] 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, (2006), 269-283.

[77] Y. Lifshits, A lower bound on the size of ϵ-free NFA corresponding to a regular expression, *Information Processing Letters*, 85, (2003), 293-299.

[78] Y. Sakuma, Y. Minamide, and A. Voronkov, Translating Regular Expression Matching into Transducers, *In Proc. of International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, (2010), 107-115.

[79] Y. Zhang, Y. Pan, and K. Chiu, Speculative p-DFAs for parallel XML parsing, *International Conference on High Performance Computing (HiPC)*, (2009), 388-397.

[80] Z. Esik and I. Simon, Modeling literal morphisms by shuffle, *Semigroup Forum*, 56(2), (1998), 225-227.