This article has been accepted for publication in a future issue of a journal/proceedings, but has not been fully edited. Content may change prior to final publication. Citation information: Belli, F., Beyazıt, M., Mutation of Directed Graphs - Corresponding Regular Expressions and Complexity of Their Generation, Proc. 11th Workshop on Descriptional Complexity of Formal Systems (DCFS 2009), Electronic Proceedings in Theoretical Computer Science (EPTCS), Vol. 3, pp. 69-77

Mutation of Directed Graphs – Corresponding Regular Expressions and Complexity of Their Generation

Fevzi Belli Mutlu Beyazıt
University of Paderborn
Faculty of Computer Science, Electrical Engineering and Mathematics
Germany
belli@adt.upb.de beyazit@adt.upb.de

Directed graphs (DG), interpreted as state transition diagrams, are traditionally used to represent finite-state automata (FSA). In the context of formal languages, both FSA and regular expressions (RE) are equivalent in that they accept and generate, respectively, type-3 (regular) languages. Based on our previous work, this paper analyzes effects of graph manipulations on corresponding RE. In this present, starting stage we assume that the DG under consideration contains no cycles. Graph manipulation is performed by deleting or inserting of nodes or arcs. Combined and/or multiple application of these basic operators enable a great variety of transformations of DG (and corresponding RE) that can be seen as mutants of the original DG (and corresponding RE). DG are popular for modeling complex systems; however they easily become intractable if the system under consideration is complex and/or large. In such situations, we propose to switch to corresponding RE in order to benefit from their compact format for modeling and algebraic operations for analysis. The results of the study are of great potential interest to mutation testing.

1 Introduction and related work

Most of model-based testing techniques operate on graphs, especially on directed graphs (DG). This has been masterly expressed by one of the testing pioneers, Beizer, as "Find a graph and cover it!" [1, 5]. The basic idea behind "graph coverage" entails generation of test cases and the selection of a minimum number of them, called "test suite", in order to cost-effectively exercise a given set of structural or functional issues of the software under test (SUT). A good test coverage increases user confidence in software artifacts, showing that the software is doing everything as it is supposed to do (*positive* testing, [4]).

For implementation-oriented, white-box testing, nodes of the DG to be covered usually represent the statements of SUT; arcs represent the sequences of those statements [10]. For specification-oriented, black-box testing, nodes of the DG may represent the behavioral events of SUT; arcs represent the sequences of those events [2].

When using a graph to model of SUT, Belli et al. propose not only to cover the DG model given, but also its complement, showing that the software is *not* doing anything it is *not* supposed to do (*negative* testing, [2, 4]). For this, the authors propose specific manipulation operators of the graph that models SUT. Negative testing approach can be seen in relationship with mutation testing, which is originally a white-box test technique [7]. Recently, Belli et al. proposed to extend mutation-testing approach to black box, model-based testing [3].

A tough problem with complex SUTs is that modeling graphs rapidly become large and thus tedious to work with. If the modeling DG can be interpreted as the transition diagram of a finite-state automaton (FSA), it might be helpful to transform the modeling DG into an algebraic format, i. e., regular expressions (RE), and work with this compact formulae instead of spacious graphs (also see [19]). Thereby,

J. Dassow, G. Pighizzini, B. Truthe (Eds.): 11th International Workshop
on Descriptional Complexity of Formal Systems (DCFS 2009)
EPTCS 3, 2009, pp. 69–77, doi:10.4204/EPTCS.3.6

well-known algorithms can be used to solve the problems concerning the transformation from DG to RE, and v. v. [9, 18, 12]. In order to extract the RE from a given DG, one may follow the steps given below:

- Convert DG to deterministic FSA (by interpreting the DG as a Moore Machine [15] and FSA as a Mealy Machine [14]).
- Convert the FSA to RE by using the widely known algorithms in the literature (also see [11]).

In addition, for the opposite chain of transformations, the following steps can be used:

- Convert RE to non-deterministic FSA (also see [13, 8]).
- Convert non-deterministic FSA to a deterministic FSA (and minimize).
- Convert the FSA to DG (similar to Mealy - Moore conversion).

Application of the basic operators, as introduced in [3], to a DG transforms it to another DG, which likely corresponds to a different RE than the original one. Contrarily, the corresponding DG of a manipulated RE differs from the DG that corresponds to the original RE. One of the main objectives of our research is to take the initial steps in order to increase the efficiency of mutation testing by determining, if possible, correspondences between DG and RE modifications. In testing literature, there are many varied constructs, such as DG, FSA, EFSA (extended FSA), ESG and state charts etc., which are used to model a SUT. Each of these graph-based representations possesses different syntax and semantics. In fact, in many cases they are presented as an extension of one another. The common arguments which can be drawn on these structures are (1) they all have (extended) RE counterparts, and (2) the more complex the SUT gets the harder they are to work with in their graphic format. [16, 18, 2, 3]

To our knowledge there is no approach which aims to manipulate the corresponding RE in order to reflect alterations of the mutation operators performed on the given DG, or v. v. However, it is worth mentioning that there are several works on the algebra of RE which enables the transformations via some defined system of rules, such as [20, 6, 17]. Taken this into account and based on DG and RE, the next section introduces the notions used in this paper, defines basic operators for graph manipulations and finally introduces the "sum of products" format for canonic representation of regular expressions. Section 3 applies those basic operators to DG and algorithmically generates their corresponding RE. Complexity of these algorithms are determined (see also the Appendix), before Section 4 concludes the paper with a summary of results already achieved and research work planned.

2 Notions used

This section briefly and semi-formally summarizes notions we need to launch the discussion in Section 3.

2.1 Directed graphs and regular expressions

Definition 1. A *directed graph* (*DG*) is the tuple (V, A) where V is a finite set of *nodes*, i. e., $V = \{v_1, \ldots, v_n\}$, and A is a finite set of *directed arcs* which are ordered pairs of elements of V, i. e., $A = \{a_1, \ldots, a_m\} \subseteq V \times V$, where each $a_i = (v_j, v_k)$ for some j, k.

Definition 2. A *regular expression* (*RE*) consists of *symbols* of an *alphabet* and is used to express a set of *strings* (or *words*), i. e., a *language*. In an operational perspective, a RE can be assumed to be a sequence of symbols $a, b, c, \ldots$ of an alphabet which can be connected by operations

- *sequence* (".", but usually no explicit operation symbol, e. g., "ab" means "b follows a"),

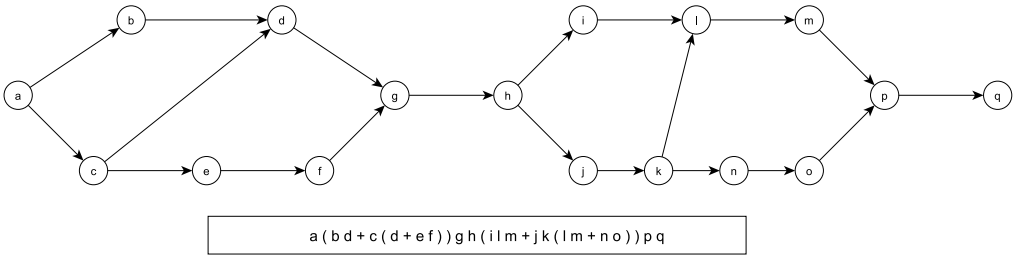

Figure 1: A sample DG and its corresponding RE

- *selection* ("+", e. g. "$a+b$" means "a or b"),
- *iteration* ("*", *Kleene's Star Operation*, e. g.,
  - "a^*" means "a will be repeated arbitrarily";
  - "a^+" means at least one occurrence of "a").

These operations are also applied on RE other than simple symbols and, as usual, *parenthesization* is used to guarantee the intended precedence and associativity.

A sample DG and its corresponding RE are given in Figure 1. In order to define a RE representation of a DG, we need to distinguish some nodes as *start* nodes and some others as *finish* nodes. In this context, the set of nodes is considered as the *alphabet* (the set of *symbols*) and the *words* (*strings*) in the language expressed by the RE are, in fact, the node sequences forming paths connecting start nodes to finish nodes in the graph. This convention has been introduced in [2] to define *event sequence graphs* (*ESG*).

2.2 Operators for manipulation of directed graphs

For manipulation of a graph, or a DG, elementary operations can be classified under two categories, *insertion* (i) and *omission* (o), and since a DG consists of nodes and edges, the manipulation operators can be specified as *node insertion* (i_n), *node omission* (o_n), *arc insertion* (i_a) and *arc omission* (o_a) operators.

Definition 3. DG manipulation operators transform a DG to another DG and defined as follows:

- *Arc insertion* operator adds a new arc (v_j, v_k), where $v_j, v_k \in V$, to the DG (V, A):
$$(v_j, v_k)i_a : (V, A) \rightarrow (V, A \cup \{(v_j, v_k)\}).$$
- *Arc omission* operator deletes an arc (v_j, v_k), where $v_j, v_k \in V$, from the DG (V, A):
$$(v_j, v_k)o_a : (V, A) \rightarrow (V, A \setminus \{(v_j, v_k)\}).$$
It is possible that some nodes are left with no ingoing and/or outgoing arcs.
- *Node insertion* operator adds a new node $v \notin V$ to the DG (V, A) together with possibly nonzero number of arcs $\{a_1, \ldots, a_k\}$ connecting this node to the remaining nodes:
$$(v, a_1, \ldots, a_k)i_n : (V, A) \rightarrow (V \cup v, A \cup \{a_1, \ldots, a_k\}).$$
- *Node omission* operator deletes a node $v \in V$ from the DG (V, A) together with all the arcs $a_1, \ldots, a_k$ A ingoing to and outgoing from the deleted node:
$$(v)o_n : (V, A) \rightarrow (V \setminus \{v\}, A \setminus \{a_1, \ldots, a_k\}).$$

Figure 2 results from the application of basic manipulation operators to the DG in Figure 1.

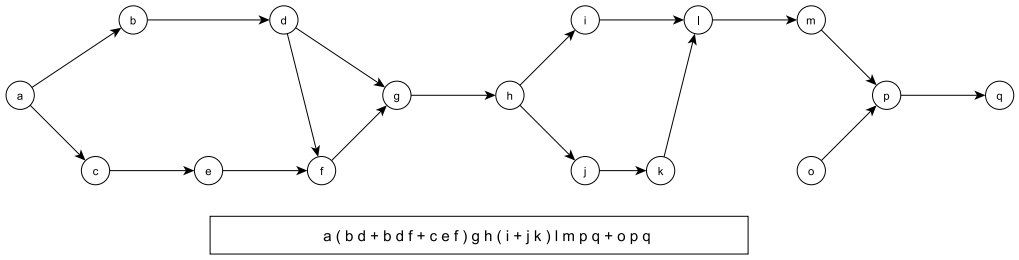

Figure 2: The manipulated DG and RE by application of $(cd)o_a(df)i_a(n)o_n$

2.3 Sum of products format for RE and auxiliary functions

In order to carry out transformation on the RE in an algorithmic way, we introduce, in analogy to Boolean Algebra, a canonical representation for RE under consideration and some auxiliary functions which operate on RE.

Definition 4. A given RE is in the *sum of products format* (*SOPF*), if it is represented as the sum of finitely many *product terms*, each of which is in one of the following forms:

- r
- R^*
- Finite concatenations of r and/or R^* (such as rR^*, R^*r and rR^*rR^*, etc.)

where r is an arbitrary finite string (formed by only the concatenation of symbols) and R is a RE in SOPF. Note that the SOPF is a very simple and straightforward format which highly disregards the compactness.

Example 5. Let the RE in Figure 1 be R, then SOPF of R is given as below

$$\begin{aligned} SOPF(R) = & abdghilmpq + abdghjklmpq + abdghjknopq + \\ & acdghilmpq + acdghjklmpq + acdghjknopq + \\ & acefghilmpq + acefghjklmpq + acefghjknopq. \end{aligned}$$ ◇

Definition 6. Let D be the DG with the set of vertices $\{v_1, \ldots, v_n\}$ and R be the corresponding RE, then we define:

- $pt(R, s)$ to be the set of product terms which contain the string s,
- $ht(P, s)$ to be the set of product terms which are the beginning (*head*) subterms, ending with the first occurrence of the string s, of the product terms in the set P, and
- $tt(P, s)$ to be the set of product terms which are the ending (*tail*) subterms, beginning with the last occurrence of the string s, of the product terms in the set P.

Example 7. Let SOPF of the RE in Figure 1 be R, then we have

- $pt(R, jkl) = \{abdghjklmpq, acdghjklmpq, acefghjklmpq\}$,
- $ht(P, f) = \{acef\}$, and
- $tt(P, gh) = \{ghilmpq, ghjklmpq, ghjknopq\}$. ◇

3 Approach: graph manipulation and effects on corresponding regular expression

As basically discussed in the introduction section, the problem, in its general form, is to expose the underlying correspondences between DG and RE manipulations. More precisely, we have following situation: Given a DG and the corresponding RE, we want to reflect the result of DG transformations stemming from applications of basic manipulation operators to the corresponding RE, and v. v. For this purpose, we assume that (1) the initial and transformed DG have no cycles and (2) all the RE are in the SOPF. Furthermore, the analysis of the effects of RE manipulation on corresponding DG, is postponed; in this stage we just focus on the effects of manipulating DG on its RE.

Under the assumptions stated above, the following subsections outline straightforward algorithms for basic manipulation of DG by transforming its corresponding RE, and, in the discussion ahead, $|P|$ and $|p|$ are defined to be upper bounds on the number of product terms and on the lengths of the product terms in the given RE. Complexity values of the auxiliary algorithms are included in Appendix and Table 1; they are necessary for the validation of the worst case time complexity results of the DG manipulation algorithms which are analyzed in the next subsections.

3.1 Arc operators

Following, omission and insertion operations are applied to arcs.

3.1.1 Arc insertion

Algorithm 1 outlines the addition of new paths connecting start and finish nodes in the DG as product terms to the given RE, during the insertion of the arc (v_i, v_j), where $v_i, v_j \in V$, to the DG. During the insertion, no product term in the RE should contain the symbol v_j before v_i. Otherwise, the operation produces a cycle.

Input: R - a RE in SOPF,
(v_i, v_j), where $v_i, v_j \in V$ - arc to be inserted
Output: R - RE is updated with the insertion of the arc (v_i, v_j) in SOPF

$A = pt(R, v_i)$ // Find the set of product terms containing v_i
$B = pt(R, v_j)$ // Find the set of product terms containing v_j
$A' = ht(A, v_i)$ // Construct the set of head subterms for A
$B' = tt(B, v_j)$ // Construct the set of tail subterms for B
$C' = A'.B'$ // Perform the set concatenation operation on A' and B'
$R = R + RE(C')$ // Add the terms in C' as product terms to R

Algorithm 1: Arc Insertion

As implied by Algorithm 1, in the insertion of the arc (v_i, v_j), the number of new product terms to be added to the RE is given by $|A'||B'|$, where $|A'|$ is the number of (distinct) head subterms leading to the node v_i from the start nodes and $|B'|$ is the number of (distinct) tail subterms leading to the finish nodes from the node v_j.

Algorithm 1 is terminating, since all the subroutines are executed in finite time. Furthermore, a straightforward calculation using the values in Table 1 shows that Algorithm 1 has the worst case time complexity $O(|P|^4|p|)$. It is possible to reduce this complexity value to $O(|P|^2|p|)$ by performing the set concatenation without filtering the duplicate product terms while constructing the set C' in $O(|P|^2|p|)$

time. These duplicate terms can be left out during the set union operation without affecting its worst case time complexity.

3.1.2 Arc omission

Omission of an arc may leave some nodes with no ingoing and/or outgoing edges. These nodes are considered as valid start and/or finish nodes respectively, because the succeeding operations may introduce new edges to such nodes. Thus, Algorithm 2 updates the given corresponding RE after the omission of the arc (v_i, v_j), where $v_i, v_j \in V$, from the DG.

Input: R - a RE in SOPF,
(v_i, v_j), where $v_i, v_j \in V$ - arc to be omitted
Output: R - RE is updated with the omission of the arc (v_i, v_j) in SOPF

$A = pt(R, v_i)$ // Find the set of product terms containing v_i
$B = pt(R, v_j)$ // Find the set of product terms containing v_j
$C = pt(R, v_i v_j)$ // Find the set of product terms containing $v_i v_j$
$A' = \emptyset$
if $A = C$ **then**
  $A' = ht(A, v_i)$ // Construct the set of head subterms for A
endif
$B' = \emptyset$
if $B = C$ **then**
  $B' = tt(B, v_j)$ // Construct the set of tail subterms for B
endif
$C' = A' \cup B'$ // Union of A' and B'
$R = R - RE(C)$ // Remove the product terms in C from R
$R = R + RE(C')$ // Add the terms in C' as product terms to R

Algorithm 2: Arc Omission

In Algorithm 2, the number of product terms to be added to and removed from the RE is given by $|A'| + |B'|$ and $|C|$, respectively, where $|A'|$ is the number of (distinct) head subterms leading to the node v_i from the start nodes, $|B'|$ is the number of (distinct) tail subterms leading to the finish nodes from the node v_j and $|C|$ is the number of (distinct) product terms containing the sequence $v_i v_j$.

The worst case time complexity of Algorithm 2 is $O(|P|^2|p|)$ (see Table 1 for the complexity of auxiliary algorithms), and it runs in finite time.

3.2 Node operators

As a next step, omission and insertion operations are applied to nodes.

3.2.1 Node insertion

Node insertion is a higher level operation when compared to arc manipulation operations, because it generally requires connecting the node to the remaining nodes. To do this, first, the inserted node is considered as a valid start and finish node. Later, the following arc insertions take place. Accordingly, Algorithm 3 can be applied to update the corresponding RE with the insertion of the node v_i together with the arcs (v_i, x_j) and (y_k, v_i), where $v_i \notin V$ and $x_j, y_k \in V$ for $j = 1, \ldots, s$ and $k = 1, \ldots, t$, to the DG.

Input: R - a RE in SOPF,
$v_i \notin V$ - node to be inserted
(v_i, x_j), where $x_j \in V$, $j = 1, \ldots, s$ - outgoing arcs to be inserted
(y_k, v_i), where $y_k \in V$, $k = 1, \ldots, t$ - ingoing arcs to be inserted
Output: R - RE is updated with the insertion of the node v_i in SOPF

$R = R + v_i$ // Add the symbol v_i as a product term to R
for each (v_i, x_j) **do**
  insert the arc (v_i, x_j) and update R // See Algorithm 1
endfor
for each (y_k, v_i) **do**
  insert the arc (y_k, v_i) and update R // See Algorithm 1
endfor
if $s \geq 1$ **or** $t \geq 1$ **then**
  $R = R - v_i$ // Remove the product term v_i from R
endif

Algorithm 3: Node Insertion

It is straightforward to note that, given the set union and arc insertion operations run in finite time, Algorithm 3 runs in finite time, and it has $O((s+t)(|P|^2|p|))$ worst case time complexity where s and t are the number of ingoing and outgoing arcs to be inserted, respectively. Note that, in a DG with no cycles, $(s+t) \leq n$ always holds and $|p|$ can be chosen to be n.

3.2.2 Node omission

Node omission entails the deletion of the node and the arcs related to it, and therefore is also a higher level operation with respect to the arc manipulation operations. For omission of a node, the node is disconnected from the rest of the graph and considered as a valid start and finish node, and later removed. Algorithm 4 shows the steps to update the corresponding RE with the omission of the node v_i (and all the arcs (v_i, x_j) and (y_k, v_i), where $x_j, y_k \in V$) from the DG.

Input: R - a RE in SOPF,
$v_i \in V$ - node to be omitted
(v_i, x_j), where $x_j \in V$, $j = 1, \ldots, s$ - outgoing arcs to be omitted
(y_k, v_i), where $y_k \in V$, $k = 1, \ldots, t$ - ingoing arcs to be omitted
Output: R - RE is updated with the omission of the node v_i in SOPF

for each (v_i, x_j) **do**
  omit the arc (v_i, x_j) and update R // See Algorithm 2
endfor
for each (y_k, v_i) **do**
  omit the arc (y_k, v_i) and update R // See Algorithm 2
endfor
$R = R - v_i$ // Remove the product term v_i from R

Algorithm 4: Node Omission

In Algorithm 4, the operations arc omission and set difference takes finite number of steps to complete. Furthermore, since the DG has no cycles, the loops are executed at most n times. Thus, the algorithm runs in finite time. In addition, in the worst case, running time complexity of Algorithm 4 is $O(k|P|^2|p|)$ where k is the total number of arcs to be omitted. Also, in a DG with no cycles, $k < n$ always holds and choosing $|p| = n$ is valid.

4 Conclusion and future work

This paper considers the effects of basic DG manipulations on the corresponding RE and outlines algorithms in order to transform the RE accordingly, where DG contains no cycles. Hence, it is an initial step to lay out the correspondence between DG and RE mutations from a practical point of view. Some of the main implications of the study, so far, can be summarized as below in two parts:

(i) Format of the RE: The size of a RE can be defined as its length, i. e., the total number of symbols and operators in the RE, and is determined by its format. The size, thus the format, of the RE has a direct effect on the efficiency of the operations. Unfortunately, SOPF is a kind of "worst-case" format where the compactness is not a concern. However, it helps to keep the algorithms straightforward and simple, and it seems easier to conserve since no additional transformations are required to preserve the format of the RE. Nevertheless, the derived complexity values should be interpreted as the "worst" of the worst case time complexity values (keeping in mind that this does not always lead to worst performance in practice).

(ii) Extent of the approach: The DG in our present paper are assumed to be free of cycles, but this does not necessarily mean that the DG models which contain cycles are completely out of the scope. One can apply different cycle omission strategies, such as traveling cycles at most a predefined number of times, in combination with the underlying semantics of the system and the indexing mechanism to update or "flatten" the DG model. Inevitably, the resulting model is only a submodel, however, in practice, there might cases where it is preferable.

On the other hand, our future work will include DG with cycles and enhance the format of the RE without sacrificing the (practical) efficiency which might stem from possible additional transformations. It is one of our concerns to improve the compactness of the RE by keeping it in another format (like perhaps product of sums format (POSF), which seems to be somehow more promising, etc.). However, it would be better and nicer to develop an approach which handles the manipulation operators in an algebraic manner without any respect to the format of the RE.

References

[1] B. Beizer (1990): *Software Testing Techniques*. Van Nostrand Reinhold.

[2] F. Belli (2001): *Finite-State Testing and Analysis of Graphical User Interfaces*. In: *Proc. of 12th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE CS Press, pp. 34–43.

[3] F. Belli, Ch. J. Budnik & E. Wong (2006): *Basic Operations for Generating Behavioral Mutants*. In: *Proc. 2nd Workshop on Mutation Analysis in conjunction with ISSRE'06*. IEEE CS, pp. 10–18.

[4] F. Belli & M. Linschulte (2008): *On 'Negative' Tests of Web Applications*. *Annals of Mathematics, Computing & Teleinformatics* 1, pp. 44–56.

[5] R. V. Binder (2000): *Testing Object-Oriented Systems*. Addison-Wesley, Boston.

[6] J. A. Brzozowski (1964): *Derivatives of Regular Expressions*. *J. ACM* 11, pp. 481–494.

[7] R. A. DeMillo, R. J. Lipton & F. G. Sayward (1978): *Hints on Test Data Selection: Help for the Practicing Programmer*. *IEEE Computer* 11, pp. 34–41.

[8] V. Geffert (2003): *Translation of binary regular expressions into nondeterministic ϵ-free automata with $O(nlogn)$ transitions*. *J. Comput. Syst. Sci.* 66, pp. 451–472.

[9] A. Gill (1962): *Introduction to the Theory of Finite-State Machines*. McGraw-Hill.

[10] S. Gossens, F. Belli, S. Beydeda & M. Dal CIN (2005): *View Graphs for Analysis and Testing of Programs at Different Abstraction Levels*. In: *Proc. of High-Assurance Systems Eng. Symp. (HASE)*. IEEE CS Press, pp. 121–13.

[11] Y. Han & D. Wood (2007): *Obtaining shorter regular expressions from finite-state automata*. Theor. Comput. Sci. 370, pp. 110–120.

[12] J. E. Hopcroft, R. Motwani & J. D. Ullman (2001): *Introduction to Automata Theory, Languages and Computation – 2nd Edition*. Addison-Wesley, Massachusetts.

[13] J. Hromkovic, S. Seibert & T. Wilke (2001): *Translating regular expressions into small ϵ-free nondeterministic finite automata*. J. Comput. Syst. Sci. 62, pp. 565–588.

[14] G. H. Mealy (1955): *A method for synthesizing sequential circuits*. Bell System. Tech. J 34, pp. 1045–1079.

[15] E. F. Moore (1956): *Gedanken experiments on sequential machines*. Automata Studies, Ann. of Math. Studies 34, pp. 129–153.

[16] J. Myhill (1957): *Finite automata and the representation of events*. Technical Report WADD TR-57-624, Dayton, OH: Wright Patterson Air Force Base.

[17] A. Salomaa (1966): *Two Complete Axiom Systems for the Algebra of Regular Events*. J. ACM 13, pp. 158–169.

[18] A. Salomaa (1969): *Theory of Automata*. Pergamon Press,Oxford, etc.

[19] A. C. Shaw (1980): *Software Specification Languages Based on Regular Expressions*. In: W. E. Riddle & R. E. Fairley, editors: Software Development Tools. Springer, Berlin, pp. 148–176.

[20] R. E. Stearns & J. Hartmanis (1963): *Regularity Preserving Modifications of Regular Expressions*. Information and Control 6, pp. 55–69.

Appendix. Some auxiliary functions and their complexity

Worst case time complexity values of some related auxiliary functions are given in Table 1. In order to interpret the complexity values correctly, note following:

- $|P|$ is an upper bound on the number of product terms in P, i. e., the number of product terms in P, and $|p|$ is an upper bound on lengths of the product terms in P, i. e., the length of the longest product term in P.
- $|p'|$ is the length of the product term p'.
- $|s|$ is the length of the string s.

Note that the sets A, B and C, and the RE R are also sets of product terms.

Table 1: Worst Case Time Complexity Values for Some Auxiliary Functions

Function	**Complexity**
Removal of a product term from a set: $P = P - p'$	$O(\|P\|(\|p\| + \|p'\|))$
Addition of a product term to a set: $P = P + p'$	$O(\|P\|(\|p\| + \|p'\|))$
Set Union: $C = A \cup B$	$O(\|A\|\|B\|(\|a\| + \|b\|))$
Set Concatenation: $C = A.B$	$O((\|A\|\|B\|)^2(\|a\| + \|b\|))$
Extraction of Tail Product Terms: $tt(P, s)$ where $\|s\| = 1, 2$	$O(\|P\|^2\|p\|)$
Extraction of Head Product Terms: $ht(P, s)$ where $\|s\| = 1, 2$	$O(\|P\|^2\|p\|)$
Extraction of Product Terms: $pt(R, s)$ where $\|s\| = 1, 2$	$O(\|P\|^2\|p\|)$