Theoretical Computer Science 48 (1986) 117-126
North Holland

NOTE

FROM REGULAR EXPRESSIONS TO DETERMINISTIC AUTOMATA

Gerard BERRY

Ecole Nationale Supérieure des Mines de Paris, Centre de Mathématiques Appliquées, Sophia-Antipolis, 06560 Valbonne France

Ravi SETHI

AT&T Bell Laboratories, Murray Hill, NY 07974, U.S.A.

Communicated by D. Perrin
Received August 1986

Abstract. The main theorem allows an elegant algorithm to be refined into an efficient one. The elegant algorithm for constructing a finite automaton from a regular expression is based on 'derivatives of' regular expressions; the efficient algorithm is based on 'marking of' regular expressions.

Derivatives of regular expressions correspond to state transitions in finite automata. When a finite automaton makes a transition under input symbol a, a leading a is stripped from the remaining input. Correspondingly, if the input string is generated by a regular expression E, then the *derivative of E by a* generates the remaining input after a leading a is stripped. Brzozowski (1964) used derivatives to construct finite automata; the state for expression E has a transition under a to the state for the derivative of E by a. This approach extends to regular expressions with new operators, including intersection and complement; however, explicit computation of derivatives can be expensive.

Marking of regular expressions yields an expression with distinct input symbols. Following McNaughton and Yamada (1960), we attach subscripts to each input symbol in an expression; $(ab+b)^*ba$ becomes $(a_1b_2+b_3)^*b_4a_5$. Conceptually, the efficient algorithm constructs an automaton for the marked expression. The marks on the transitions are then erased, resulting in a nondeterministic automaton for the original unmarked expression. This approach works for the usual operations of union, concatenation, and iteration; however, intersection and complement cannot be handled because marking and unmarking do not preserve the languages generated by regular expressions with these operators.

1. Introduction

We study two well-known algorithms for constructing a finite automaton from a regular expression. An elegant algorithm due to Brzozowski [6] will be developed into an efficient algorithm based on McNaughton and Yamada [11]. Brzozowski's algorithm is easily seen to be correct, it accommodates additional operators like intersection and complement, and it has recently served as a starting point for compiling communicating processes in the Esterel programming language [3] into automata [4]. The efficient algorithm is used in fast pattern matchers like *egrep*, distributed as part of the UNIX[1] operating system. A brief account of *grep* [14]

[1] UNIX is a trademark of AT&T Bell Laboratories.

and its cousins, including *egrep*, appears in [1]. A version of *egrep*'s algorithm is described in [2, Section 3.9].

The syntax of regular expressions over a set Σ of input symbols is (a is a typical symbol):

$$E ::= 0 \mid 1 \mid a \mid E+E \mid E\cdot E \mid E^*.$$

$L(E)$ denotes the language generated by a regular expression E. $L(0)$ is the empty set and $L(1)$ is the set consisting of the empty string ϵ. Note that 0 and 1 are not input symbols; they represent the sets of strings $\emptyset$ and $\{\epsilon\}$. $L(E+F)$ is the union of $L(E)$ and $L(F)$. $L(E\cdot F)$ consists of strings formed by concatenating a string in $L(E)$ with a string in $L(F)$. $L(E^*)$ consists of strings formed by concatenating zero or more strings from $L(E)$; $L(E^*)$ includes the empty string ϵ.

We write $E \equiv F$ if $L(E) = L(F)$. The following properties of regular expressions will be used without fanfare:

$$0+E \equiv E \equiv E+0,$$

$$0\cdot E \equiv 0 \equiv E\cdot 0, \qquad 1\cdot E \equiv E \equiv E\cdot 1.$$

Using Brzozowski's notation, $\delta(E)$ stands for 1 if $L(E)$ contains the empty string; otherwise, $\delta(E)$ stands for 0. It can be computed from the structure of E:

$$\delta(0) = 0, \qquad \delta(1) = 1, \qquad \delta(a) = 0,$$

$$\delta(E+F) = \delta(E) + \delta(F), \qquad \delta(E\cdot F) = \delta(E)\cdot\delta(F),$$

$$\delta(E^*) = 1.$$

Thus, $\delta(E)\cdot F$ equals F if the empty string is in $L(E)$; otherwise, $\delta(E)\cdot F$ equals 0. Furthermore, $E + \delta(E) \equiv E$, because $E+0 \equiv E$ and, when ϵ is in $L(E)$, $E+1 \equiv E$.

2. Derivatives of regular expressions

Brzozowski's algorithm [6] is based on the notion of a 'derivative' of a regular expression E with respect to an input symbol a, written $a^{-1}E$. Informally, if leading a's are stripped from strings in $L(E)$ that start with a, we get the strings generated by $a^{-1}E$. The derivative of $aba+bb$ by a is ba.

Definition 2.1. Given a regular expression E and a symbol a, the *derivative of E by a*, written $a^{-1}E$, is defined by

$$a^{-1} = 0, \qquad a^{-1}0 = 0,$$

$$a^{-1}a = 1, \qquad a^{-1}b = 0 \quad \text{for } b \neq a,$$

$$a^{-1}(E+F) = a^{-1}E + a^{-1}F,$$

$$a^{-1}(E\cdot F) = a^{-1}E\cdot F + \delta(E)\cdot a^{-1}F,$$

$$a^{-1}(E^*) = a^{-1}E\cdot E^*.$$

Within expressions, a^{-1} is treated as a prefix operator with higher precedence than $+, \cdot$, and $*$.

Remark 2.2. Additional operators like intersection and complement can be handled by adding rules of the form:

$$a^{-1}(E \cap F) = a^{-1}E \cap a^{-1}F, \qquad a^{-1}(E - F) = a^{-1}E - a^{-1}F.$$

Informally, it does not matter if leading a's are stripped before or after the operations are performed. For example,

$$a^{-1}(ab^* \cap a) \equiv 1 \equiv (a^{-1}(ab^*)) \cap (a^{-1}a),$$

$$a^{-1}(ab^* - a) \equiv b^* - 1 \equiv (a^{-1}(ab^*)) - (a^{-1}a)$$

States in the constructed automaton correspond to regular expressions. There is a transition under a from the state for E to the state for $a^{-1}E$. The transition under a from the state for $aba + bb$ is to the state for ba; the subsequent transition under b is to the state for a. It is easier to talk about sequences of transitions if the notion of derivatives is generalized from symbols to strings. It is easier to write $(ab)^{-1}E$ than $b^{-1}(a^{-1}E)$. More significantly, the next definition allows us to write $w^{-1}E$, where the string w is any member of a set of strings.

Definition 2.3. The extension from symbols to the derivative of E by a string is defined by

$$\epsilon^{-1}E = E, \qquad (wa)^{-1}E = a^{-1}(w^{-1}E).$$

Within expressions, w^{-1} is treated as a prefix operator with higher precedence than $+, \cdot$, and $*$.

Automata will be constructed by explicitly computing derivatives, then derivatives of derivatives, and so on, as needed. Convergence is guaranteed by the following result from [6].

Proposition 2.4. *The set of derivatives of a regular expression is finite, modulo associativity, commutativity, and idempotence of* $+$; *that is, the set* $\{F \mid \exists w\colon F = w^{-1}E\}$ *has a finite number of equivalence classes.*

Without associativity, commutativity, and idempotence of $+$, duplicate subexpressions would cause successive derivatives of $E = a^*(aa)^*$ by $a, aa, \ldots$ to be distinct. These properties allow a sum of expressions to be treated as a set of expressions, thereby removing duplicates.

Algorithm 2.5 (Brzozowski [6]) (*Construction of a deterministic automaton D accepting* $L(E)$).

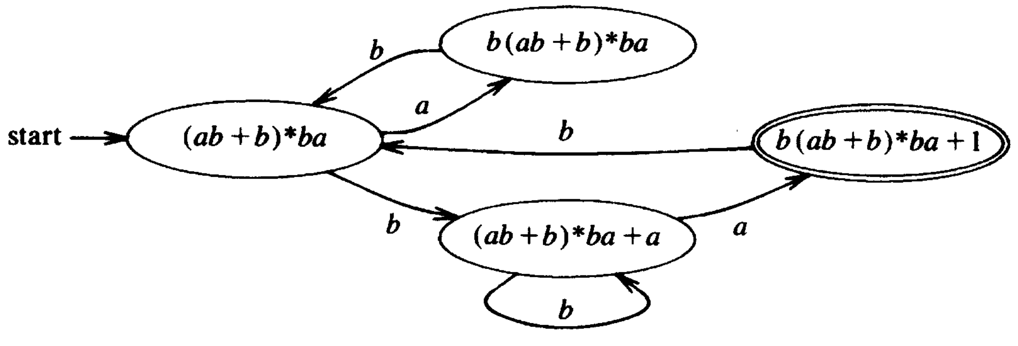


Fig. 1. Automaton accepting $(ab+b)^*ba$.

(1) The states of D are the distinct derivatives $w^{-1}E$, for all strings w. Proposition 2.4 ensures convergence of this step.

(2) Construct a transition under a from state p to state q if and only if p is for derivative $w^{-1}E$, for some w, and q is for $(wa)^{-1}E$.

(3) The state for E is the start state. A state is an accepting state if and only if it is for a derivative $w^{-1}E$, for some w, and $\delta(w^{-1}E)=1$; that is, the empty string is in $L(w^{-1}E)$.

Algorithm 2.5 constructs the automaton in Fig. 1. Each state of the automaton is for a derivative of $E=(ab+b)^*ba$; the state for $w^{-1}E$ has a transition under a to the state for $(wa)^{-1}E$.

3. Regular expressions with distinct symbols

Following McNaughton and Yamada [11], we mark all input symbols in a regular expression to make them distinct. The marks are written as subscripts; a marked version of $(ab+b)^*ba$ is $(a_1b_2+b_3)^*b_4a_5$, where a_1 and a_5 are treated as different symbols.

The construction of a deterministic automaton D from a regular expression E is outlined in Fig. 2. E' is formed by marking all symbols in E to make them distinct. Suppose that an automaton M' accepts $L(E')$. We show that unmarking the symbols labeling the transitions of M' yields a nondeterministic automaton M. D can be obtained from M by applying the standard subset construction [2, 13].

Remark 3.1. The approach of Fig. 2 does not extend to regular expressions with intersection and complement operators. Although $(ab^*)\cap a\equiv a$, we get $(a_1b_2{}^*)\cap a_3\equiv 0$ because a_1 and a_3 are distinct. Similarly, $(ab^*)-a\equiv abb^*$, but $(a_1b_2{}^*)-a_3\equiv a_1b_2{}^*$. The unmarking homomorphism does not commute with intersection and relative complementation of languages.

$$\begin{tikzcd}
E \arrow[r, "\text{mark symbols}"'] & E' \arrow[r] & M' \arrow[r, "\text{unmark symbols}"'] & M \arrow[r, "\text{subset construction}"'] & D
\end{tikzcd}$$

Fig. 2. From a regular expression E to a deterministic finite automaton D.

The approach of Fig. 2 works for union, concatenation, and iteration because unmarking commutes with these operations on languages.

Proposition 3.2. *Let E' be the regular expression obtained from E by marking all symbols to make them distinct. If M' is an automaton accepting $L(E')$, then M, a nondeterministic automaton accepting $L(E)$, is obtained by unmarking all the symbols labeling the transitions in M'.*

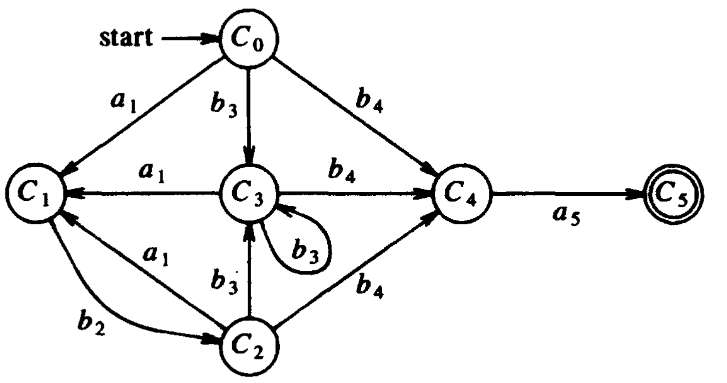
Proof. Let *unmark* map marked symbols to their original unmarked form. We write $L(A)$ for the language accepted by automaton A.

A structural induction on E establishes that $L(E) = \mathit{unmark}(L(E'))$. For the basis step, note that marking has no effect on 0 and 1, and that unmarking recovers a symbol a from its marked counterpart. The inductive step, consisting of cases for the operators $+, \cdot$, and $*$, is not shown.

For each transition of M' on a marked symbol, there is a corresponding nondeterministic transition of M on the unmarked symbol, so M accepts a string w if M' accepts a string w' such that $w = \mathit{unmark}(w')$. The converse holds as well, so $L(M) = \mathit{unmark}(L(M'))$.

By construction, $L(E') = L(M')$. The result $L(E) = L(M)$ follows by transitivity. □

The main theorem in this section allows each symbol in a marked expression to be viewed as a state of an automaton. Figure 3 contains a motivating example for the theorem. Each state of the automaton in Fig. 3 is labeled with a symbol representing a derivative of the expression $(a_1b_2 + b_3)^*b_4a_5$. Furthermore, as in automata constructed by Algorithm 2.5, the state for C has a transition under a marked symbol to a state for the derivative of C by that symbol. By construction, all the transitions entering a state are labeled with the same symbol; see, for example, the transitions labeled b_3 into the state for C_3. All strings that drive the automaton from the start state into C_3 must therefore be of the form wb_3, for some w. By construction, if a string wb_3 drives the automaton into a state for expression C_3, then C_3 must equal the derivative by wb_3 of the starting expression C_0. Theorem



$$\begin{aligned}
C_0 &= (a_1b_2+b_3)^*b_4a_5\\
C_1 &= b_2(a_1b_2+b_3)^*b_4a_5\\
C_2 &= (a_1b_2+b_3)^*b_4a_5\\
C_3 &= (a_1b_2+b_3)^*b_4a_5\\
C_4 &= a_5\\
C_5 &= 1
\end{aligned}$$

Fig. 3. Automaton for $(a_1b_2 + b_3)^*b_4a_5$.

3.4 will show that the equivalence between C_3 and $(wb_3)^{-1}C_0$, for all w, is no accident; it follows from the distinctness of all symbols in the expression C_0.

The proof of the main theorem uses the following lemma.

Lemma 3.3. *Given any symbol a, for all strings w, $(wa)^{-1}(E^*)$ is equivalent to a sum of subterms chosen from the set $\{(va)^{-1}E\cdot E^* \mid wa = uva\}$.*

Proof. By induction on the length of w. The basis, length 0, follows from the definition of derivatives because $a^{-1}(E^*) = a^{-1}E\cdot E^*$. For the inductive step, suppose $w = xb$. By definition,

$$(xba)^{-1}(E^*) = a^{-1}((xb)^{-1}(E^*)).$$

From the inductive hypothesis, $(xb)^{-1}(E^*)$ is equivalent to a sum of subterms chosen from $\{(zb)^{-1}E\cdot E^* \mid xb = yzb\}$. The operator a^{-1} distributes across a sum:

$$a^{-1}\left(\sum_{x=yz} (zb)^{-1}E\cdot E^*\right) = \sum_{x=yz} a^{-1}((zb)^{-1}E\cdot E^*).$$

Applying the rule for $\cdot$ in Definition 2.1, the right-hand side yields

$$\sum_{x=yz} (zba)^{-1}E\cdot E^* + \sum_{x=yz} \delta((zb)^{-1}E)\cdot a^{-1}(E^*). \tag{1}$$

Each subterm of the form $\delta((zb)^{-1}E)$ is either 0 or 1. Therefore, the second summation in (1) equals $a^{-1}(E^*) = a^{-1}E\cdot E^*$ if it does not equal 0. All terms in (1) are therefore of the form $(va)^{-1}E\cdot E^*$, where $wa = uva = xba$, so the lemma holds. □

Theorem 3.4 (Main Theorem). *Let all symbols in E be distinct. Given any symbol a, for all strings w, $(wa)^{-1}E$ is either 0 or unique modulo associativity, commutativity, and idempotence of $+$.*

Proof. By structural induction on E. If E is either 0 or 1, then all derivatives are 0. Otherwise, if it is a symbol, then the only possible values for its derivatives are 0 and 1. In the remaining cases, we use the structure of E to expand $(wa)^{-1}E$.

Case 1: $E = E_1 + E_2$. It follows from the definition of derivatives that

$$(wa)^{-1}(E_1 + E_2) = (wa)^{-1} + (wa)^{-1}E_2. \tag{2}$$

Since all symbols in E are distinct, if a is in E_1, then $(wa)^{-1}E_2 = 0$; otherwise, a is in E_2 and $(wa)^{-1}E_1 = 0$. This case follows from the inductive hypothesis applied to the remaining term in (2).

Case 2: $E = E_1\cdot E_2$. An auxiliary induction on the length of w establishes

$$(wa)^{-1}(E_1\cdot E_2) = (wa)^{-1}E_1\cdot E_2 + \sum_{wa=uva} \delta(u^{-1}E_1)\cdot(va)^{-1}E_2. \tag{3}$$

If a is in E_1, then only the first term on the right-hand side of (3) survives and the inductive hypothesis applies to it. Otherwise, a is in E_2 and terms of the form $\delta(u^{-1}E_1)\cdot(va)^{-1}E_2$ are left. Recall that $\delta(u^{-1}E_1)$ is either 0 or 1, for all u, and from the inductive hypothesis, all subterms of the form $(va)^{-1}E_2$ are equivalent to either 0 or some fixed term F, so their sum is also either 0 or equivalent to that fixed term F.

Case 3: $E = E_1{}^*$. From Lemma 3.3, $(wa)^{-1}(E^*)$ is equivalent to a sum of terms chosen from the set $\{(va)^{-1}E\cdot E^* \mid wa = uva\}$; a complete characterization of $(wa)^{-1}(E^*)$ is not needed. From the inductive hypothesis, each nonzero subterm $(va)^{-1}E$ must therefore be equivalent to some fixed term F, so any sum of subterms of the form $(va)^{-1}E\cdot E^*$ is equivalent to $F\cdot E^*$. This final case holds because, being such a sum, $(wa)^{-1}(E^*)$ is equivalent to $F\cdot E^*$ if it is not 0. $\square$

The automaton in Fig. 3 is constructed by a refinement of Algorithm 2.5. The new algorithm works with a specific set of derivatives, corresponding to the distinct symbols in a marked expression. Since this set is fixed in advance, convergence of the new algorithm is immediate; correctness carries over from Algorithm 2.5.

Definition 3.5. Let all symbols in E be distinct. For all symbols a in E, a *continuation of a in E* is any expression $(wa)^{-1}E \neq 0$. By structural induction on E, such an expression must exist; by the above theorem, all such expressions are equivalent. We therefore speak of 'the' continuation of a in E to refer to some expression in the equivalence class.

Algorithm 3.6 (*Construction of a deterministic automaton M' from a marked expression E'*).

(1) M' has a state for the continuation of each marked symbol in E'.

(2) Construct a transition under a from state p to the state for the continuation of a if and only if p is for some continuation C and C can generate a string with a leading a.

(3) The state for the entire expression E' is the start state. A state is an accepting state if and only if it is for a continuation C and $\delta(C) = 1$.

The states of automaton in Fig. 3 are labeled with continuations of marked input symbols in the expression $(a_1b_2 + b_3)^*b_4a_5$. For all i, $1 \leq i \leq 5$, C_i is the continuation of the symbol marked i, and C_0 represents the entire expression. Although the automata constructed by the new algorithm can have more states—compare Fig. 3 and Fig. 1—the new algorithm compensates by not checking expressions for equivalence. The next section shows that the continuations in Fig. 3 need not be computed explicitly either.

4. A fast algorithm

Algorithm 3.6 can be improved. Since each continuation is uniquely determined by an input symbol, a regular expression with n marked symbols will lead to an automaton with $n+1$ states—a start state and a state for each symbol. Instead of complete continuations, the second step of the algorithm needs only the set of leading symbols in strings generated by the continuations. These sets are related to 'follow sets' defined below.

Definition 4.1

$$\mathit{first}(E)=\{a \mid av\in L(E)\}, \qquad \mathit{follow}_E(a)=\{b \mid uabv\in L(E)\}.$$

Expressions of the form $E!$, where ! is a new endmarker symbol, are used below to avoid special cases in the computation of follow sets for the 'last' symbols that can be generated by E. If a is such a last symbol, then $\mathit{follow}_{E!}(a)$ will contain !.

Proposition 4.2. *Let all symbols in E be distinct and, for all a, let C_a be the continuation of a in E. Then,* $\forall a$: $\mathit{first}(C_a!)=\mathit{follow}_{E!}(a)$.

Proof. Brzozowski's [6] observation that every regular expression E can be represented as an infinite sum of terms of the form $w\cdot w^{-1}E$, formalizes the idea that the derivative of $w^{-1}E$ is formed by stripping a prefix w from strings generated by E. Restating the result for the derivatives of E by strings of the form wa, we find that

$$E\equiv\delta(E)=\sum_{wa} wa\cdot(wa)^{-1}E.$$

From Theorem 3.4, $(wa)^{-1}E$ is either 0 or C_a, so

$$E!\equiv\delta(E)\cdot!+\sum_{(wa)^{-1}E\neq 0} wa\cdot C_a\cdot! \tag{4}$$

For all a, symbol a appears just before C_a in the right-hand side of (4), so $\mathit{first}(C_a!)$ is a subset of $\mathit{follow}_{E!}(a)$.

For the converse, suppose b is in $\mathit{follow}_{E!}(a)$. Then, for some u and v, $uabv!\in L(E!)$, and from (4) $uabv!\in L(ua\cdot C_a\cdot!)$. Hence, b must be in $\mathit{first}(C_a!)$. □

An algorithm for computing follow sets is given in [2, Section 3.9]. A related algorithm is illustrated in Fig. 4. The figure contains a syntax tree for the expression

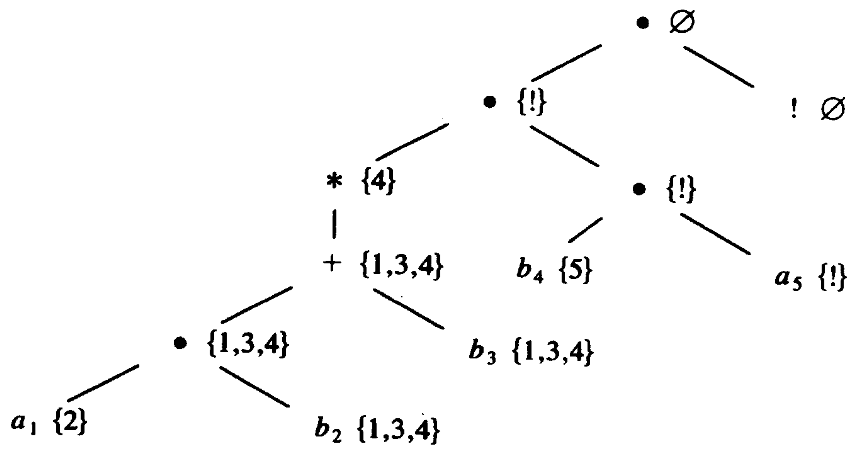


Fig. 4. Follow sets for subexpressions of $(a_1b_2+b_3)^*b_4a_5!$.

$(a_1b_2+b_3)^*b_4a_5!$, where ! is an endmarker. For clarity, follow sets are written in terms of integer subscripts on the marked symbols. The notion of follow sets is generalized from symbols to subexpressions represented by nodes in the syntax tree. To the right of each node appears the set of symbols that can follow the last symbol generated by the subexpression at the node. Alternatively, the set can be thought of as an attribute inherited by the node. Proceeding top down from the root with $\emptyset$, the set at a node is accumulated until the leaf for a symbol a is reached with $\mathit{follow}_{E!}(a)$.

The rules used for computing the sets in Fig. 4 are summarized in the next proposition.

Proposition 4.3. *Let E be a regular expression with distinct symbols.* F, *defined by the rules below, is such that* $\mathbf{F}(E, \{!\})$ *yields a set of pairs of the form* $\langle a, \mathit{follow}_{E!}(a)\rangle$. *The rules are*:

$$\mathbf{F}(E_1+E_2, S) = \mathbf{F}(E_1, S) \cup \mathbf{F}(E_2, S),$$

$$\mathbf{F}(E_1 \cdot E_2, S) = \mathbf{F}(E_1, \mathit{first}(E_2 \cup \delta(E_2) \cdot S) \cup \mathbf{F}(E_2, S),$$

$$\mathbf{F}(E_1^*, S) = \mathbf{F}(E_1, \mathit{first}(E_1) \cup S),$$

$$\mathbf{F}(a, S) = \langle a, S\rangle, \qquad \mathbf{F}(1, S) = \emptyset, \qquad \mathbf{F}(0, S) = \emptyset.$$

Clément and Kahn [8] have adapted the above rules to construct a Typol [7] implementation. The Typol specification is built up of sequents of the form $S \vdash E_1 : \langle \mathit{first}, \delta, \mathbf{F}\rangle$, where $\mathit{first} = \mathit{first}(E_1)$, $\delta = \delta(E_1)$, and $\mathbf{F} = \mathbf{F}(E_1, S)$. Their rule for $E_1 \cdot E_2$ is

$$\frac{\mathit{first}(E_2) \cup \delta(E_2) \cdot S \vdash E_1 : \langle \mathit{first}_1, \delta_1, \mathbf{F}_1\rangle \qquad S \vdash E_2 : \langle \mathit{first}_2, \delta_2, \mathbf{F}_2\rangle}{S \vdash E_1 \cdot E_2 : \langle \mathit{first}_1 \cup \delta_1 \cdot \delta_2, \mathbf{F}_1 \cup \mathbf{F}_2\rangle}.$$

Once *first* and *follow* sets are computed, Algorithm 3.6 is improved into Algorithm 4.4 below.

Algorithm 4.4 (*Fast construction of a deterministic automaton M′ from a marked expression E′*).

(1) M' has a start state plus a state for each marked symbol a_i in E'.

(2) Construct a transition from the start state to the state for a_i if and only if $a_i \in \mathit{first}(E')$; construct a transition from the state for b_j to the state for a_i if and only if $a_i \in \mathit{follow}_{E'!}(b_j)$.

(3) The start state is an accepting state if and only if $\delta(E') = 1$; the state for a_i is an accepting state if and only if $! \in \mathit{follow}_{E'!}(a_i)$.

5. Discussion

We no longer need to choose between the simpler and more general approach based on derivatives of regular expressions and the efficient approach based on marking and *follow* sets. Theorem 3.4 and Proposition 4.2 relate the two approaches, so the derivative approach can be used as the starting point for constructing automata. These results also verify the correctness of algorithms based on marking and *follow* sets.

Regular expressions with union, concatenation, and iteration operators suffice for specifying patterns in strings [1, 14]. Additional operators are used in applications like protocol validation [5, 9] and communication between processes [3, 10, 12]. The idea of marking and unmarking symbols in regular expressions does not always extend to additional operators; see the examples involving intersection and complement in Remark 3.1. The idea of *follow* sets can, however, be used even if marking cannot.

Acknowledgments

We thank Al Aho, Dominique Clément, and Gilles Kahn for helpful comments.

References

[1] A.V. Aho, Pattern matching in strings, in: R.V. Book, ed., *Formal Language Theory* (Academic Press, New York, 1980 325–347.

[2] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools* (Addison-Wesley, Reading, MA 1986).

[3] G. Berry and L. Cosserat, The Esterel synchronous programming language and its mathematical semantics, in: S.D. Brookes, A.W. Roscoe and G. Winskel, eds., *Seminar in Concurrency*, Lecture Notes in Computer Science **197** (Springer, Berlin, 1985).

[4] G. Berry, P. Couronne and G. Gonthier, The ESTEREL v2.1 System Manuals, Tech. Rept., Ecole des Mines/INRIA, 1986.

[5] G.V. Bochmann, Communication protocols and error recovery procedures, *ACM Operating Systems Review* **9**(3) (1975) 45–50.

[6] J.A. Brzozowski, Derivatives of regular expressions, *J. ACM* **11**(4) (1964) 481–494.

[7] D. Clement, J. Despeyroux, T. Despeyroux, L. Hascoet and G. Kahn, Natural semantics on the computer, Tech. Rept. No. 416, INRIA, Sophia-Antipolis, 1985.

[8] D. Clement and G. Kahn, Personal communication, February 1986.

[9] G.J. Holzmann, A theory for protocol validation, *IEEE Trans. Comput.* **C-31**(8) (1982) 730–738.

[10] J. Katzenelson and R.P. Kurshan, S/R: A language for specifying protocols and other coordinating processes, *5th Phoenix Conf. on Computer Communications* (1986) 286–292.

[11] R. McNauthton and H. Yamada, Regular expressions and state graphs for automata, *IRE Trans. on Electronic Comput.* **EC-9**(1) (1960) 38–47.

[12] R. Milner, A complete inference system for a class of regular behaviours, *J. Comput. System Sci.* **28** (1984) 439–466.

[13] M.O. Rabin and D. Scott, Finite automata and their decision problems, *IBM J. Res. Develop.* **3**(2) (1959) 114–125.

[14] K. Thompson, Regular expression search algorithm, *Comm. ACM* **11**(6) (1968) 419–422.