On the correctness of some algorithms to generate finite automata for regular expressions

Eindhoven University of Technology

Department of Mathematics and Computing Science

On the Correctness of some Algorithms to Generate Finite Automata for Regular Expressions

by

H.M.M. ten Eikelder and H.P.J. van Geldrop

93/32

Computing Science Note 93/32
Eindhoven, September 1993

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. M. Philips
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

ON THE CORRECTNESS OF SOME ALGORITHMS TO GENERATE FINITE AUTOMATA FOR REGULAR EXPRESSIONS

H.M.M ten Eikelder & H.P.J. van Geldrop
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract

We discuss the method given by Glushkov, McNaughton-Yamada and Berry-Sethi and a related method given by Aho, Sethi and Ullmann to generate a deterministic finite automaton accepting the language of a given regular expression. For both methods a formal description and a simple correctness proof is given.

Contents

1 Introduction

There are several methods to construct a deterministic finite automaton that accepts the language described by a given regular expression. In this note we discuss two methods. The first method[1] is described by Glushkov [Glu, theorem 16], McNaughton and Yamada [MY, page 44] and Berry and Sethi [BS, Algorithm 4.4]. A strongly related method has been given by Aho, Sethi and Ullman [ASU,Algorithm 3.5]. In this note we give a simple correctness proof for both methods. It is based on a simple property, from which the correctness of both methods can easily be derived. The proof uses neither techniques like derivatives of regular expressions (see [Br]) nor algebra's of automata as in [Wa].

Both methods consist in fact of two steps: (i) first the construction of a non-deterministic automaton corresponding to the given regular expression is described, (ii) then this automaton is converted into a deterministic one. Since step (ii) is a standard construction, we shall mainly focus on step (i). In Section 2, regular expressions and various related notions, such as the language $\mathcal{L}(e)$ corresponding to a given regular expression e, will be defined. In Section 3, an alternative "language" $\mathcal{K}(e)$ corresponding to e is defined. In general $\mathcal{L}(e) \subseteq \mathcal{K}(e)$. However, we shall formulate a condition $C(e)$ which implies that the languages are equal. This property is the basis for the correctness proof of the two constructions. A special kind of finite automata is introduced in Section 4. We describe two mappings of regular expressions to finite automata. One of these mappings corresponds to the McNaughton-Yamada, Glushkov and Berry-Sethi construction, the other corresponds to the Aho-Sethi-Ullman construction. It is easily proved that both mappings, applied to an arbitrary regular expression e, yield automata that accept the language $\mathcal{K}(e)$. Hence, if the condition $C(e)$ holds, we have automata accepting $\mathcal{L}(e)$. The restriction that this construction can only be applied to the class of regular expressions satisfying $C(e)$ will be removed in Section 5. There we show how for an arbitrary regular expression e a "marked version" e' can be constructed such that $C(e')$ holds. Moreover the automata for $\mathcal{L}(e')$ can easily be adapted such that they accept $\mathcal{L}(e)$. In general this process yields non-deterministic automata. The conversion to deterministic automata (step (ii) above) is in fact a standard construction and is described in Section 6. For automata obtained with the Aho-Sethi-Ullman mapping we show that the combination of step (i) and (ii) finally leads to the algorithm described in [ASU].

2 Preliminaries

In this section we give the definitions of various well-known notions in the field of regular languages. In this note we always assume that V is a (finite) alphabet.

Definition 1 (RE_V, **regular expressions over** V) *The set RE_V of regular expressions over V is the smallest set X satisfying the following rules. For $a \in V$, $e, f \in X$:*

$$\begin{array}{ll} \varepsilon & \in X \\ a & \in X \\ (e \mid f) & \in X \\ (e \cdot f) & \in X \\ e^* & \in X \end{array}$$

□

As usual we shall reduce the number of parentheses by: (i) giving priorities to the operators: $prio(*) > prio(\cdot) > prio(|)$, (ii) using the associativity of the operators $\cdot$ and $|$, (iii) not writing outermost parentheses.

[1] The methods described by Glushkov, McNaughton-Yamada and Berry-Sethi are very similar. However, the used markings (see section 5) are different.

Definition 2 ($\mathcal{L}$, language defined by regular expression) *The mapping $\mathcal{L}_V : RE_V \rightarrow \mathcal{P}(V^*)$ is defined recursively as follows. For all $a \in V, e, f \in RE_V$:*

$$\begin{aligned}
\mathcal{L}_V(\varepsilon) &= \{\varepsilon\} \\
\mathcal{L}_V(a) &= \{a\} \\
\mathcal{L}_V(e \mid f) &= \mathcal{L}_V(e) \cup \mathcal{L}_V(f) \\
\mathcal{L}_V(e \cdot f) &= \mathcal{L}_V(e)\mathcal{L}_V(f) \\
\mathcal{L}_V(e^*) &= (\mathcal{L}_V(e))^*
\end{aligned}$$

□

Language (and string) concatenation is denoted by juxtaposition. $\mathcal{L}_V(e)$ will be called the language defined by the regular expression e. If the alphabet V is obvious, the mapping $\mathcal{L}_V$ will be written as $\mathcal{L}$.

Definition 3 (*Null*) *The predicate Null on RE_V is defined recursively as follows. For all $a \in V, e, f \in RE_V$:*

$$\begin{aligned}
Null(\varepsilon) &= true \\
Null(a) &= false \\
Null(e \mid f) &= Null(e) \vee Null(f) \\
Null(e \cdot f) &= Null(e) \wedge Null(f) \\
Null(e^*) &= true
\end{aligned}$$

□

Definition 4 (*First*) *The mapping $First : RE_V \rightarrow \mathcal{P}(V)$ is defined recursively as follows. For all $a \in V, e, f \in RE_V$:*

$$\begin{aligned}
First(\varepsilon) &= \emptyset \\
First(a) &= \{a\} \\
First(e \mid f) &= First(e) \cup First(f) \\
First(e \cdot f) &= \begin{cases} First(e) & if\ \neg Null(e) \\ First(e) \cup First(f) & otherwise \end{cases} \\
First(e^*) &= First(e)
\end{aligned}$$

□

Definition 5 (*Last*) *The mapping $Last : RE_V \rightarrow \mathcal{P}(V)$ is defined recursively as follows. For all $a \in V, e, f \in RE_V$:*

$$\begin{aligned}
Last(\varepsilon) &= \emptyset \\
Last(a) &= \{a\} \\
Last(e \mid f) &= Last(e) \cup Last(f) \\
Last(e \cdot f) &= \begin{cases} Last(f) & if\ \neg Null(f) \\ Last(e) \cup Last(f) & otherwise \end{cases} \\
Last(e^*) &= Last(e)
\end{aligned}$$

□

Definition 6 (*Follow*) *The mapping $Follow : RE_V \rightarrow \mathcal{P}(V \times V)$ is defined recursively as follows. For all $a \in V, e, f \in RE_V$:*

$$\begin{aligned}
Follow(\varepsilon) &= \emptyset \\
Follow(a) &= \emptyset
\end{aligned}$$

$$
\begin{aligned}
Follow(e \mid f) &= Follow(e) \cup Follow(f) \\
Follow(e \cdot f) &= Follow(e) \cup Follow(f) \cup (Last(e) \times First(f)) \\
Follow(e^*) &= Follow(e) \cup (Last(e) \times First(e))
\end{aligned}
$$

□

It is easily seen that $Null(e) = \varepsilon \in \mathcal{L}(e)$. The sets $First(e)$ and $Last(e)$ consist of all symbols that can appear as the first, respectively last, element of a string from $\mathcal{L}(e)$. Furthermore $(a, b) \in Follow(e)$ if $\mathcal{L}(e)$ contains a string that has ab as a substring. Summarizing, if $w \in V^*$, then

$$
\begin{aligned}
& w \in \mathcal{L}(e) \\
(7) \qquad & \Rightarrow \\
& (w = \varepsilon \wedge Null(e)) \vee \\
& (w_1 \in First(e) \wedge w_n \in Last(e) \wedge (\forall i : 1 \leq i < n : (w_i, w_{i+1}) \in Follow(e)))
\end{aligned}
$$

where $n = \mid w \mid$. Note that in general the reverse implication does not hold, take for instance $e = aa$ and $w = aaa$. In the next section we shall impose an additional condition on e which ensures equivalence in (7).

3 A basic property

Here we formulate a condition which ensures equivalence in (7). This equivalence will be the basis for our explanation of the Glushkov-McNaughton-Yamada and Berry-Sethi algorithms. First we give some definitions.

Definition 8 ($\underline{\text{in}}$,#) *The mapping* $\underline{\text{in}} : V \times RE_V \rightarrow \mathbb{B}$ *(written in infix notation) is defined recursively as follows. For all* $a, b \in V, e, f \in RE_V$:

$$
\begin{aligned}
a \ \underline{\text{in}} \ \varepsilon &= false \\
a \ \underline{\text{in}} \ b &= (a = b) \\
a \ \underline{\text{in}} \ e \cdot f &= a \ \underline{\text{in}} \ e \vee a \ \underline{\text{in}} \ f \\
a \ \underline{\text{in}} \ e \mid f &= a \ \underline{\text{in}} \ e \vee a \ \underline{\text{in}} \ f \\
a \ \underline{\text{in}} \ e^* &= a \ \underline{\text{in}} \ e
\end{aligned}
$$

The mapping $\# : V \times RE_V \rightarrow \mathbb{N}$ *is defined recursively as follows. For all* $a, b \in V, e, f \in RE_V$:

$$
\begin{aligned}
\#(a, \varepsilon) &= 0 \\
\#(a, b) &= \begin{cases} 0 & if \quad a \neq b \\ 1 & if \quad a = b \end{cases} \\
\#(a, e \cdot f) &= \#(a, e) + \#(a, f) \\
\#(a, e \mid f) &= \#(a, e) + \#(a, f) \\
\#(a, e^*) &= \#(a, e)
\end{aligned}
$$

□

So a $\underline{\text{in}}$ e means that the alphabet symbol a occurs somewhere in the regular expression e and $\#(a, e)$ denotes the number of occurences of a in e.

Definition 9 (C, S) *The mapping* $C : RE_V \rightarrow \mathbb{B}$ *is defined as follows. For all* $e \in RE_V$:

$$C(e) = (\forall a \in V :: \#(a, e) \leq 1)$$

The mapping $S : RE_V \rightarrow \mathcal{P}(V)$ *is defined as follows. For all* $e \in RE_V$:

$$S(e) = \{a \in V \mid a \underline{\text{in}}\ e\}$$

□

So $C(e)$ means that each alphabet symbol appears at most once in the regular expression e . For instance $C(a \cdot b \mid c)$ holds but $C(a \cdot b \mid a)$ does not hold. Furthermore $S(e)$ is the set of alphabet symbols that actually appear in e, e.g. $S(a \cdot b \mid c) = \{a, b, c\}$. Using definition 8 the following relations are easily proved.

(10) $$C(e_1 \mid e_2) = C(e_1) \wedge C(e_2) \wedge (S(e_1) \cap S(e_2) = \emptyset)$$

(11) $$C(e_1 \cdot e_2) = C(e_1) \wedge C(e_2) \wedge (S(e_1) \cap S(e_2) = \emptyset)$$

(12) $$C(e^*) = C(e)$$

To formulate the main theorem of this section we give the following definition[2].

Definition 13 $(P, \mathcal{K})$ *The mapping* $P : V^* \times RE_V \rightarrow \mathbb{B}$ *is defined as follows. For all* $w \in V^*$ *and* $e \in RE_V$:

$$\begin{array}{l} P(w, e) \\ = \\ (w = \varepsilon \wedge Null(e)) \vee \\ (w_1 \in First(e) \wedge w_n \in Last(e) \wedge (\forall i : 1 \leq i < n : (w_i, w_{i+1}) \in Follow(e))) \end{array}$$

where $n = \mid w \mid$.
The mapping $\mathcal{K} : RE_V \rightarrow \mathcal{P}(V^*)$ *is defined as follows:*

$$\mathcal{K}(e) = \{w \in V^* \mid P(w, e)\}$$

□

Clearly (see (7)) we have that for all regular expressions e

$$\mathcal{L}(e) \subseteq \mathcal{K}(e)$$

In the remaining part of this section we shall formulate conditions which imply that both languages are equal. The equality is then proved using induction with respect to the regular expression. The following lemma's correspond to the various cases in that proof.

Lemma 14 $\mathcal{K}(\varepsilon) = \{\varepsilon\}$

[2]Note that $P(w, e)$ is equal to the right hand side of (7).

Proof: Trivially $P(w, \varepsilon) = (w = \varepsilon)$.
□

Lemma 15 *For all symbols* $a \in V$:
$\mathcal{K}(a) = \{a\}$

Proof: Trivially $P(w, a) = (w = a)$ for all $a \in V$.
□

Lemma 16 *For all* $e_1, e_2 \in RE_V$ *with* $S(e_1) \cap S(e_2) = \emptyset$:
$\mathcal{K}(e_1 \mid e_2) = \mathcal{K}(e_1) \cup \mathcal{K}(e_2)$

Proof: It is sufficient to prove that for all $w \in V^*$:

$$P(w, e_1 \mid e_2) = P(w, e_1) \vee P(w, e_2)$$

$\Leftarrow$: Trivially for all $e_1, e_2 \in RE_V : P(w, e_1) \vee P(w, e_2) \Rightarrow P(w, e_1 \mid e_2)$
$\Rightarrow$: For $w = \varepsilon$ this follows immediately from the definition of *Null*. Using $S(e_1) \cap S(e_2) = \emptyset$ we observe that for $w \neq \varepsilon : P(w, e_1 \mid e_2) \wedge w_1 \in S(e_i) \Rightarrow w \in S(e_i)^*$ for $i = 1, 2$. Hence, for $w \neq \varepsilon : P(w, e_1 \mid e_2) \wedge w_1 \in S(e_i) \Rightarrow P(w, e_i)$ for $i = 1, 2$.
□

Lemma 17 *For all* $e_1, e_2 \in RE_V$ *with* $S(e_1) \cap S(e_2) = \emptyset$:
$\mathcal{K}(e_1 \cdot e_2) = \mathcal{K}(e_1)\mathcal{K}(e_2)$

Proof: It is sufficient to prove that for all $w \in V^*$:

$$P(w, e_1 \cdot e_2) = (\exists u, v \in V^* : w = uv : P(u, e_1) \wedge P(v, e_2))$$

$\Leftarrow$: Trivially for all $e_1, e_2 \in RE_V : P(u, e_1) \wedge P(v, e_2) \Rightarrow P(uv, e_1 \cdot e_2)$
$\Rightarrow$: This implication is proved by using the observation that, since $S(e_1) \cap S(e_2) = \emptyset$, we have $P(w, e_1 \cdot e_2) \Rightarrow w \in S(e_1)^* S(e_2)^*$.
□

Lemma 18 *For all* $e \in RE_V$:
$\mathcal{K}(e^*) = \mathcal{K}(e)^*$

Proof: It is sufficient to prove that for all $w \in V^*$:

$$P(w, e^*) = w \in \mathcal{K}(e)^*$$

$\Leftarrow$: Trivially $P(\varepsilon, e^*)$ and $P(u, e) \wedge P(v, e^*) \Rightarrow P(uv, e^*)$.
$\Rightarrow$: Assume $P(w, e^*)$ holds. Using $\mathit{Follow}(e^*) = \mathit{Follow}(e) \cup (\mathit{Last}(e) \times \mathit{First}(e))$ we conclude that there exists a (possibly empty) index set $I \subseteq \{1, \ldots, \mid w \mid -1\}$ such that

$$(\forall i : 1 \leq i < \mid w \mid \wedge i \notin I : (w_i, w_{i+1}) \in \mathit{Follow}(e)) \wedge$$
$$(\forall i : i \in I : (w_i, w_{i+1}) \in (\mathit{Last}(e) \times \mathit{First}(e)) \setminus \mathit{Follow}(e))$$

Hence w can be written as a concatenation of $l =\mid I \mid +1$ strings, say $w = v^1 \ldots v^l$ such that $P(v^i, e)$ holds for all $i \in \{1, \ldots, l\}$. This means that $w \in \mathcal{K}(e)^l \subseteq \mathcal{K}(e)^*$.

□

Next we state the main theorem of this section.

Theorem 19 *Let $e \in RE_V$ such that $C(e)$ holds. Then*

$$\mathcal{L}(e) = \mathcal{K}(e)$$

Proof: Using the lemma's 14 - 18 the induction proof is easily given.

□

4 Automata for a restricted class of regular expressions

The result proved in theorem 19 can be used to construct two types of finite automata accepting the language of a given regular expression. In the first instance this method can only be applied to regular expressions for which condition C holds. In the next section we shall dispose of this restriction. First we give a definition of the used type of finite automaton.

Definition 20 (*SNFA*) *A special finite automaton (SNFA) is a 5-tuple (Q, W, Δ, S, F) with :*

Q a finite set of states,
W a finite alphabet,
Δ, the transition relation, is a subset of $Q \times W \times Q$,
$S \subseteq Q$ is the set of start states,
$F \subseteq Q$ is the set of final states.

□

The characteristic properties of an *SNFA* automaton are that: (i) it has a *set* of start states and (ii) there are no ϵ-transitions. The set of all *SNFA* automata over an alphabet V will be denoted by $\mathcal{SNFA}_V$. In cases where the alphabet is obvious, the subscript V will be omitted.

Definition 21 ($\mathbb{L}_V$) *The language $\mathbb{L}_V(M)$ accepted by an SNFA $M = (Q, W, \Delta, S, F)$ is defined as:*

$$\mathbb{L}_V(M)$$
$$=$$
$$\{w \in V^* \mid (\exists q_0, \ldots, q_{|w|} \in Q :: q_0 \in S \wedge q_{|w|} \in F \wedge (\forall i : 0 \leq i <\mid w \mid : (q_i, w_{i+1}, q_{i+1}) \in \Delta))\ \}$$

□

Again the subscript V will be omitted when obvious.

Next we define two mappings from regular expressions to *SNFA*'s.

Definition 22 (Glushkov McNaughton-Yamada Berry-Sethi mapping) *The mapping $\mathcal{M}_G : RE_V \rightarrow \mathcal{SNFA}$ is defined by $\mathcal{M}_G(e) = (Q, V, \Delta, \{@\}, F)$ where:*

$@ \notin V$
$Q = S(e) \cup \{@\}$
$F = Last(@ \cdot e)$

$\Delta \subseteq Q \times V \times Q$ *is such that*
$(p, a, q) \in \Delta \;=\; (p, q) \in \mathit{Follow}(@ \cdot e) \wedge a = q$

□

This method of constructing a finite automaton is part of a construction that has already been described by McNaughton and Yamada [MY] ,Glushkov [Glu] and Berry-Sethi[BS]. Note that $\mathcal{M}_G(e)$ is always a deterministic automaton because $(p, a, q_1) \in \Delta \wedge (p, a, q_2) \in \Delta \Rightarrow q_1 = q_2$.

Definition 23 (Aho-Sethi-Ullman mapping) *The mapping* $\mathcal{M}_A : RE_V \rightarrow \mathcal{SNFA}$ *is defined by* $\mathcal{M}_A(e) = (Q, V, \Delta, S, \{@\})$ *where:*

$@ \notin V$
$Q = S(e) \cup \{@\}$
$S = \mathit{First}(e \cdot @)$
$\Delta \subseteq Q \times V \times Q$ *is such that*
$(p, a, q) \in \Delta \;=\; (p, q) \in \mathit{Follow}(e \cdot @) \wedge a = p$

□

This construction can be found in [ASU]. In general $\mathcal{M}_A(e)$ will be a non-deterministic automaton[3]. Next we describe the languages accepted by these automata.

Theorem 24 *For all* $e \in RE_V$:

$$\mathbb{L}(\mathcal{M}_G(e)) = \mathcal{K}(e)$$

Proof: Let $e \in RE_V$ and let $M = \mathcal{M}_G(e) = (Q, V, \Delta, \{@\}, F)$. We consider two cases. For strings $w \in V^*$ with length $n = \mid w \mid > 0$:

$w \in \mathbb{L}(\mathcal{M})$
$=$ { definition 21 }
$(\exists q_0, \ldots, q_n \in Q :: q_0 \in S \wedge q_n \in F \wedge (\forall i : 0 \leq i < n : (q_i, w_{i+1}, q_{i+1}) \in \Delta))$
$=$ { definition 22 }
$(\exists q_0, \ldots, q_n \in Q :: q_0 = @ \wedge q_n \in \mathit{Last}(@ \cdot e) \wedge$
$(\forall i : 0 \leq i < n : w_{i+1} = q_{i+1} \wedge (q_i, q_{i+1}) \in (\{@\} \times \mathit{First}(e)) \cup \mathit{Follow}(e))$
$)$
$=$ { $n > 0$ }
$(@, w_1) \in (\{@\} \times \mathit{First}(e)) \cup \mathit{Follow}(e) \wedge w_n \in \mathit{Last}(@ \cdot e) \wedge$
$(\forall i : 0 < i < n : (w_i, w_{i+1}) \in (\{@\} \times \mathit{First}(e)) \cup \mathit{Follow}(e))$
$=$ { $@ \notin V$ }
$w_1 \in \mathit{First}(e) \wedge w_n \in \mathit{Last}(e) \wedge$
$(\forall i : 1 \leq i < n : (w_i, w_{i+1}) \in \mathit{Follow}(e))$
$=$ { definition 13 }
$w \in \mathcal{K}(e)$

For the empty string:

$\varepsilon \in \mathbb{L}(\mathcal{M})$
$=$ { definition $\mathcal{M}$ }
$@ \in \mathit{Last}(@ \cdot e)$
$=$ { $@ \notin V$ }

[3]Note that the reverse automaton, i.e. the automaton obtained by reversing all transition arrows and interchanging initial and final states, is deterministic.

$$
\begin{array}{ll}
& \mathit{Null}(e) \\
= & \quad \{ \text{ definition13 } \} \\
& \varepsilon \in \mathcal{K}(e)
\end{array}
$$

□

Theorem 25 *For all* $e \in RE_V$:

$$\mathbb{L}(\mathcal{M}_A(e)) = \mathcal{K}(e)$$

Proof: Similarly to the proof of theorem 24.
□

Theorem 26 *Let* $e \in RE_V$ *such that* $C(e)$ *holds. Then* :

$$\mathbb{L}(\mathcal{M}_G(e)) = \mathcal{L}(e)$$
$$\mathbb{L}(\mathcal{M}_A(e)) = \mathcal{L}(e)$$

Proof: Trivial using the theorems 19 and 24 respectively 25.
□

Hence we have shown, that if C holds for a regular expression e, the automata $\mathcal{M}_G(e)$ and $\mathcal{M}_A(e)$ accept the language corresponding to e. Note that $\mathcal{M}_G(e)$ is a deterministic automaton. In the next section we shall construct automata for the case that $C(e)$ does not hold. In general that will unfortunately lead to nondeterministic automata.

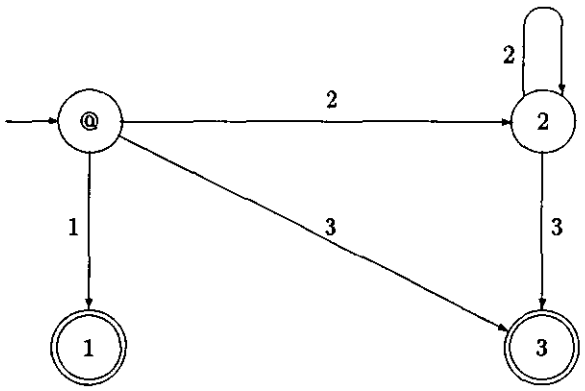
Example 27
Consider the regular expression $e = 1 \mid 2^* \cdot 3$. Then

$$S(e) = \{1, 2, 3\}$$
$$\mathit{Last}(@ \cdot e) = \{1, 3\}$$
$$\mathit{Follow}(@ \cdot e) = \{(@, 1), (@, 2), (@, 3), (2, 2), (2, 3)\}$$

The corresponding automaton $\mathcal{M}_G(e) = (Q, V, \Delta, \{@\}, F)$ is given by

$$Q = S(e) \cup \{@\} = \{1, 2, 3, @\}$$
$$F = \mathit{Last}(@ \cdot e) = \{1, 3\}$$
$$\Delta = \{(p, q, q) \mid (p, q) \in \mathit{Follow}(@ \cdot e)\} = \{(@, 1, 1), (@, 2, 2), (@, 3, 3), (2, 2, 2), (2, 3, 3)\}$$

The automaton has the following graphical representation[4].



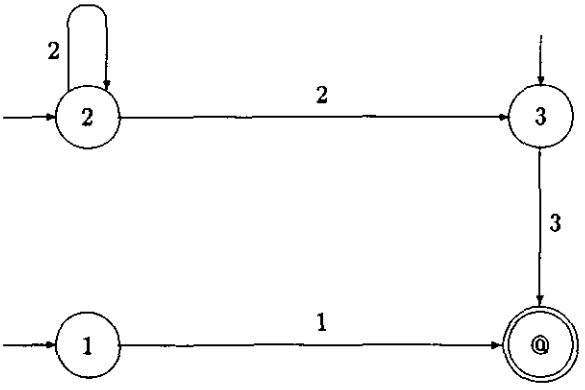
Note that this is indeed a deterministic automaton.
Furthermore

$$First(e \cdot @) = \{1, 2, 3\}$$
$$Follow(e \cdot @) = \{(1, @), (2, 2), (2, 3), (3, @)\}$$

Hence the automaton $\mathcal{M}_A(e) = (Q, V, \Delta, S, \{@\})$ is given by

$$Q = S(e) \cup \{@\} = \{1, 2, 3, @\}$$
$$S = First(e \cdot @) = \{1, 2, 3\}$$
$$\Delta = \{(p, p, q) \mid (p, q \in Follow(e \cdot @)\} = \{(1, 1, @), (2, 2, 2), (2, 2, 3), (3, 3, @)\}$$

with the following graphical representation.



Note that this is not a deterministic automaton. Since condition $C(e)$ holds, we obtain from theorem 26 that both automata given above accept the language $\mathcal{L}(e)$.

□

As already suggested by the example above, there is a simple relation between the automata obtained via the Glushkov McNaughton-Yamada Berry-Sethi mapping and the Aho-Sethi-Ullman mapping. To make this remark more explicit, we define the mappings

$Rev_M : \mathcal{SNFA} \rightarrow \mathcal{SNFA}$ by

[4] Start states will be depicted with an unlabeled incoming arrow and final states will be depicted by two concentric circles.

$$Rev_M((Q, V, \Delta, S, F)) = (Q, V, \Delta', F, S))$$
$$\text{with } (p, a, q) \in \Delta = (q, a, p) \in \Delta'$$

and

$$Rev_e : RE_V \rightarrow RE_V \text{ by}$$
$$Rev_e(\varepsilon) = \varepsilon$$
$$Rev_e(a) = a \text{ for all } a \in V$$
$$Rev_e(e_1 \mid e_2) = Rev_e(e_2) \mid Rev_e(e_1)$$
$$Rev_e(e_1 \cdot e_2) = Rev_e(e_2) \cdot Rev_e(e_1)$$
$$Rev_e(e^*) = Rev_e(e)^*$$

So Rev_M reverses *SNFA*'s and Rev_e reverses regular expressions. Then, for all e, the following relations hold:

$$Rev_M(\mathcal{M}_G(e)) = \mathcal{M}_A(Rev_e(e))$$
$$Rev_M(\mathcal{M}_A(e)) = \mathcal{M}_G(Rev_e(e))$$

Both relations are easily proved using some simple properties of *Null First Last* and *Follow*. In fact this property could also be used to prove the correctness of one of the automata from the correctness of the other one.

5 Automata for arbitrary regular expressions

Next we consider an arbitrary regular expression $e \in RE_V$. In general $C(e)$ will not hold. However, we can construct a related alphabet V' and regular expression $e' \in RE_{V'}$ such that $C(e')$ holds. Then the automaton accepting $\mathcal{L}_{V'}(e')$ can easily be adapted to an automaton accepting the original language $\mathcal{L}_V(e)$.

Definition 28 (Marking) *A marking with respect to a regular expression e over an alphabet V is a triple $< V', e', unmark >$ such that:*

- *V' is an alphabet*
- *$e' \in RE_{V'}$*
- *$C(e')$ holds*
- *$unmark : V' \rightarrow V$ is such that $e = unmark_1(e')$*

where $unmark_1 : RE_{V'} \rightarrow RE_V$ is defined as the unique homomorphic extension[5] of $unmark$ to $RE_{V'}$.
□

In the sequel of this section we assume that $< V', e', unmark >$ is a marking with respect $e \in RE_V$. Then, since $C(e')$ holds, the mappings described in the previous section yield the automata $\mathcal{M}_G(e')$ and $\mathcal{M}_A(e')$, both accepting $\mathcal{L}(e')$. Next we show that these automata can be "unmarked", thus yielding automata accepting $\mathcal{L}(e)$. To achieve this goal we give the following definitions

Definition 29 (Unmarkings) *The mappings $unmark_2 : V'^* \rightarrow V^*$, $unmark_3 : \mathcal{P}(V'^*) \rightarrow \mathcal{P}(V^*)$ and $unmark_4 : \mathcal{SNFA}_{V'} \rightarrow \mathcal{SNFA}_V$ are defined as follows:*

$$unmark_2(\varepsilon) = \varepsilon$$
$$unmark_2(a) = unmark(a) \quad \textit{for all } a \in V'$$

[5] i.e. $unmark_1(\varepsilon) = \varepsilon$, $unmark_1(a) = unmark(a)$, $unmark_1(e_1 \mid e_2) = unmark_1(e_1) \mid unmark_1(e_2)$, $unmark_1(e_1 \cdot e_2) = unmark_1(e_1) \cdot unmark_1(e_2)$ and $unmark_1(f^*) = (unmark_1(f))^*$.

$$unmark_2(xy) = unmark_2(x)unmark_2(y) \quad \textit{for all } x, y \in V'^*$$

$$unmark_3(L) = \{unmark_2(x) \mid x \in L\}$$

$$unmark_4((Q, V', \Delta', S, F)) = (Q, V, \Delta, S, F) \quad \textit{with}$$
$$\Delta = \{(p, unmark(a), q) \mid (p, a, q) \in \Delta'\}$$

So the mapping $unmark : V' \rightarrow V$ gives rise to similar "unmarkings" for regular expressions, strings, languages and finite automata[6].

Lemma 30 $unmark_3 \circ \mathcal{L}_{V'} = \mathcal{L}_V \circ unmark_1$

Proof: It is easily shown using induction that for all $e'' \in RE_{V'}$: $unmark_3(\mathcal{L}_{V'}(e'')) = \mathcal{L}_V(unmark_1(e''))$.
□

Lemma 31 $unmark_3 \circ \mathbb{L}_{V'} = \mathbb{L}_V \circ unmark_4$

Proof: Let $M = (Q, V', \Delta', S, F)$ be an *SNFA* over V'. Then $unmark_4(M) = (Q, V, \Delta, S, F)$ with $\Delta = \{(p, unmark(a), q) \mid (p, a, q) \in \Delta'\}$. We have to show that $unmark_3(\mathbb{L}_{V'}(M)) = \mathbb{L}_V(unmark_4(M))$. Let $w \in V^*$ with $n = \mid w \mid$. Then

$$\begin{array}{ll}
& w \in \mathbb{L}_V(unmark_4(M)) \\
= & \quad \{ \text{ definition 21 } \} \\
& (\exists q_0, \ldots, q_n \in Q :: q_0 \in S \wedge q_{|w|} \in F \wedge (\forall i : 0 \leq i < n : (q_i, w_{i+1}, q_{i+1}) \in \Delta)) \\
= & \quad \{ (p, a, q) \in \Delta = (\exists a' \in V' : a = unmark(a') : (p, a', q) \in \Delta') \} \\
& (\exists q_0, \ldots, q_n \in Q :: \\
& \quad q_0 \in S \wedge q_n \in F \wedge (\forall i : 0 \leq i < n : (\exists a' \in V' : w_{i+1} = unmark(a') : (q_i, a', q_{i+1}) \in \Delta')) \\
&) \\
= & \quad \{ \text{ interchanging quantifications } \} \\
& (\exists q_0, \ldots, q_n \in Q :: \\
& \quad q_0 \in S \wedge q_n \in F \wedge (\exists v \in V'^* : w = unmark_2(v) : (\forall i : 0 \leq i < n : (q_i, v_{i+1}, q_{i+1}) \in \Delta')) \\
&) \\
= & \quad \{ \text{ interchanging quantifications } \} \\
& (\exists v \in V'^* : w = unmark_2(v) : \\
& (\exists q_0, \ldots, q_n \in Q :: q_0 \in S \wedge q_n \in F \wedge (\forall i : 0 \leq i < n : (q_i, v_{i+1}, q_{i+1}) \in \Delta')) \\
&) \\
= & \quad \{ \text{ definition 21 } \} \\
& (\exists v \in V'^* : w = unmark_2(v) : v \in \mathbb{L}_{V'}(M)) \\
& = \\
& w \in unmark_3(\mathbb{L}_{V'}(M))
\end{array}$$

□

Recall that we assume in this section that $< V', e', unmark >$ is a marking of the (arbitrary) regular expression e over V. The following theorem gives finite automata accepting $\mathcal{L}(e)$.

[6] The sets RE_V, $\mathcal{P}(V)$ and $\mathcal{SNFA}$ can all be given a Σ-algebra structure, see for instance [Wa]. Then various mappings can be considered as Σ-homomorphisms and in some proofs we could use the initiality of the algebra RE_V. However, we feel that in this simple case introducing that machinery is not useful.

Theorem 32 *Both automata* $unmark_4(\mathcal{M}_G(e'))$ *and* $unmark_4(\mathcal{M}_A(e'))$ *accept the language* $\mathcal{L}(e)$.

Proof:

$$
\begin{aligned}
& \mathbb{L}_V(unmark_4(\mathcal{M}_G(e'))) \\
= \quad & \{ \text{ lemma 31 } \} \\
& unmark_3(\mathbb{L}_{V'}(\mathcal{M}_G(e'))) \\
= \quad & \{ C(e'), \text{ theorem 26 } \} \\
& unmark_3(\mathcal{L}_{V'}(e')) \\
= \quad & \{ \text{ lemma 30 } \} \\
& \mathcal{L}_V(unmark_1(e')) \\
= \quad & \{ unmark_1(e') = e \ \} \\
& \mathcal{L}_V(e)
\end{aligned}
$$

The proof for the automaton $unmark_4(\mathcal{M}_A(e'))$ proceeds similarly.

□

So far we have assumed that $< V', e', unmark >$ was a marking of e. It is easily seen that such markings exist. We give some examples

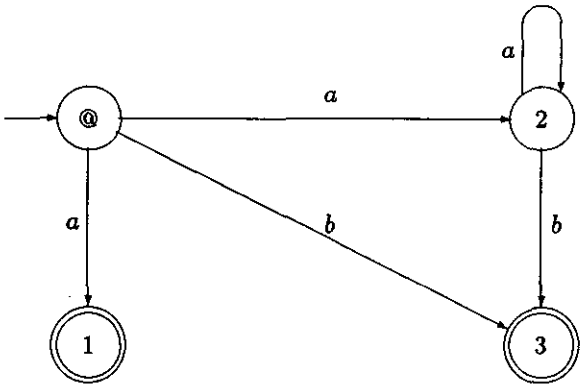
- In $e \in RE_V$ each occurrence of an alphabet symbol, say a, is replaced by the pair $< a,$ *position of this occurrence of* a *in* $e >$. Then $V' = V \times \{1, ..., n\}$ where n is the number of occurrences of alphabet symbols in e, and the mapping $unmark : V' \rightarrow V$ is projection on the first coordinate.
  For instance the expression $e = a \cdot (a \mid b)^* \cdot b$ over $V = \{a, b\}$ is mapped onto $e' =< a, 1 > \cdot (< a, 2 > | < b, 3 >)^* \cdot < b, 4 >$ over $V' = V \times \{1, \ldots, 4\}$. This marking, with the second pair element written as a subscript, is used by Glushkov [Glu] and by Berry and Sethi [BS].

- In $e \in RE_V$ each occurrence of an alphabet symbol, say a, is replaced by the position of this occurrence of a in e. Then $V' = \{1, \ldots, n\}$ where n is the number of occurrences of alphabet symbols and the mapping $unmark : V' \rightarrow V$ is defined by

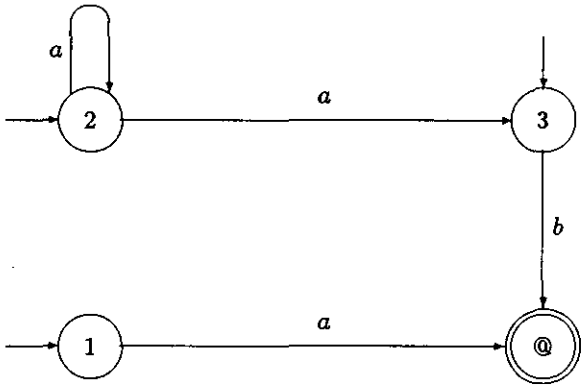
  $$unmark(k) \ = \ \text{"}symbol\ at\ position\ k\ in\ e\text{"}$$

  For instance the expression $e = a \cdot (a \mid b)^* \cdot b$ over $V = \{a, b\}$ is mapped onto $e' = 1 \cdot (2 \mid 3)^* \cdot 4$ over $V' = \{1, \ldots, 4\}$. This marking is used in [ASU], algorithm 3.5.

Example
Consider the regular expression $e = a \mid a^* \cdot b$. As the symbol a appears twice, it does not satisfy the condition $C(e)$. Following the second method above $< \{1, 2, 3\}, e', unmark >$ with $e' = 1 \mid 2^* \cdot 3$ and $unmark(1) = a, unmark(2) = a, unmark(3) = b$ is a marking of e. Note that e' equals the regular expression used in example 27. Hence the automata $\mathcal{M}_G(e')$ and $\mathcal{M}_A(e')$ are equal to the automata given in example 27. After unmarking we obtain the automaton $unmark_4(\mathcal{M}_G(e'))$ with graphical representation



and the automaton $unmark_4(\mathcal{M}_A(e'))$ with graphical representation



Theorem 32 implies that both automata accept the language $\mathcal{L}(e)$.
□

6 Deterministic automata

The mapping from regular expressions to automata described in the sections 4 and 5 consists of three steps:
(a) mark the regular expression e to obtain a marked version e',
(b) apply one of the mappings $\mathcal{M}_G$ or $\mathcal{M}_A$, as described in section 4, to e', thus obtaining an *SNFA* accepting $\mathcal{L}(e')$,
(c) unmark the obtained *SNFA* to obtain an *SNFA* accepting $\mathcal{L}(e)$.
The intermediate automaton $\mathcal{M}_G(e)$ is deterministic, while, in general, the intermediate automaton $\mathcal{M}_A(e)$ is not deterministic. However, due to the unmarking process we will generally end up with a non-deterministic automaton[7]. Since we ultimately want a *deterministic* automaton accepting $\mathcal{L}(e)$, a fourth step has to be added: transform the automaton obtained in step (c) into a deterministic one. The process of transforming a non-deterministic automaton into an equivalent deterministic automaton is well-known; it is usually denoted by the term "subset construction". The set of states of the resulting deterministic automaton is the powerset of the set of states of the original automaton. Hence this leads to a rather large automaton. In most cases this automaton contains states that cannot be reached from the start state. Eliminating these unreachable states may lead to a smaller but equivalent (deterministic) automaton. Here we will use the subset construction, followed by

[7] If *unmark* is not injective, the mapping $unmark_4$ can map a deterministic *SNFA* into a non-deterministic one.

the elimination of unreachable states. For this combination we will use the term "subset-reachable construction". We now give the various formal definitions.

Definition 33 (*DFA*) *A deterministic finite automaton* (*DFA*) *is a 5-tuple* (Q, V, δ, s, Ac) *with* :

Q *a finite set of states,*
V *a finite alphabet,*
δ, *the transition mapping, is a mapping* $Q \times V \rightarrow Q$,
$s \in Q$ *is the start state,*
$Ac \subseteq Q$ *is the set of final states.*

□

The set of all *DFA*'s will be denoted by $\mathcal{DFA}$.

Definition 34 *The language* $\mathbb{L}'(M)$ *accepted by a DFA* $M = (Q, V, \delta, s, Ac)$ *is defined as:*

$$\mathbb{L}'(\mathcal{M}) = \{w \in V^* \mid \delta^*(s, w) \in F\}$$
where the mapping $\delta^* : Q \times V^* \rightarrow Q$ *is defined by*
$\delta^*(q, \varepsilon) = q$
$\delta^*(q, aw) = \delta^*(\delta(q, a), w)$

□

Definition 35 (Subset construction) *The mapping* $Subset : \mathcal{SNFA} \rightarrow \mathcal{DFA}$ *is defined by:*

$$Subset((Q, V, \Delta, S, F)) = (\mathcal{P}(Q), V, \delta, \{S\}, F_1)$$
where
$\delta(R, a) = \{q \in Q \mid (\exists r \in R :: (r, a, q) \in \Delta)\}$
$F_1 = \{R \in \mathcal{P}(Q) \mid R \cap F \neq \emptyset\}$

Due to the absence of ε-transitions the subset construction for *SNFA*'s as described above, is somewhat simpler than the version for more general nondeterministic automata. It is a standard result that the deterministic automaton obtained by the subset construction accepts the same language as the original automaton, i.e.

(36) $\quad \mathbb{L}' \circ Subset = \mathbb{L}$

Definition 37 (Elimination of unreachable states) *The mapping* $Reachable : \mathcal{DFA} \rightarrow \mathcal{DFA}$ *is defined by:*

$$Reachable((Q, V, \delta, s, F)) = (Q_1, V, \delta_1, s, F_1)$$
with
$Q_1 = \{\delta^*(s, w) \mid w \in V^*\}$
$\delta_1 = \delta \downarrow_{Q_1 \times V}$
$F_1 = F \cap Q_1$
where δ^* *is the mapping introduced in definition 34*

Trivially the elimination of unreachable states does not change the language accepted by the automaton, i.e.

(38) $\mathbb{L}' \circ \mathit{Reachable} = \mathbb{L}'$

The composition $\mathit{Reachable} \circ \mathit{Subset}$ corresponds to the subset-reachable construction mentioned above. From (36) and (38) we obtain that

(39) $\mathbb{L}' \circ \mathit{Reachable} \circ \mathit{Subset} = \mathbb{L}$

The following standard theorem describes an algorithm for the subset-reachable construction.

Theorem 40 (Algorithm for subset-reachable construction) *For all SNFA's* $M = (Q, V, \Delta, S, F)$:

$\mathit{Reachable} \circ \mathit{Subset}((Q, V, \Delta, S, F)) = (D, V, \delta, s, Ac)$
where D, δ, s, Ac *are computed by the following algorithm:*

var $Z, G : \mathcal{P}(\mathcal{P}(Q))$; $U, T : \mathcal{P}(Q)$; $a : V$
| $Z := \emptyset$; $G := \{S\}$
; **do** $G \neq \emptyset \rightarrow$
  let $T \in G$
  ; $Z := Z \cup \{T\}$; $G := G \setminus \{T\}$
  ; **for all** $a \in V$ **do**
    $U := \{q \in Q \mid (\exists p \in T :: (p, a, q) \in \Delta)\}$
    ; **if** $U \notin G \cup Z \rightarrow$ $G := G \cup \{U\}$
    [] $U \in G \cup Z \rightarrow$ **skip**
    fi
    ; $\delta(T, a) := U$
  od
od
; $D := Z$; $s := \{S\}$; $Ac := \{U \in D \mid U \cap F \neq \emptyset\}$

□

Next we apply the subset-reachable construction to an *SNFA* obtained by step (a)-(c) above. So let e be an arbitrary regular expression over an alphabet V and let $< V', e', \mathit{unmark} >$ be a marking of e. If in step (b) the Aho-Sethi-Ullman version is chosen we obtain the following automaton (see definitions 23 and 29):

$\mathit{unmark}_4(\mathcal{M}_A(e')) = (Q, V, \Delta, S, \{@\})$
where

$Q = S(e') \cup \{@\}$
$\Delta = \{(p, \mathit{unmark}(a), q) \mid (p, q) \in \mathit{Follow}(e' \cdot @) \wedge a = p\}$
$S = \mathit{First}(e' \cdot @)$

Applying the subset-reachable construction to this *SNFA* yields:

$\mathit{Subset} \circ \mathit{Reachable}(\mathit{unmark}_4(\mathcal{M}_A(e'))) = (D, V, \delta, s, Ac)$
where D, δ, s, Ac are computed by the following algorithm:

var $Z, G : \mathcal{P}(\mathcal{P}(Q))$; $U, T : \mathcal{P}(Q)$; $a : V$
| $Z := \emptyset$; $G := \{\mathit{First}(e' \cdot @)\}$
; **do** $G \neq \emptyset \rightarrow$
  let $T \in G$

$$
\begin{array}{l}
\quad ;\ Z := Z \cup \{T\}\ ;\ G := G \setminus \{T\} \\
\quad ;\ \textbf{for all } a \in V \textbf{ do} \\
\qquad U := \{q \in Q \mid (\exists p \in T :: (p,q) \in \mathit{Follow}(e' \cdot @) \wedge a = \mathit{unmark}(p))\} \\
\qquad ;\ \textbf{if } U \notin G \cup Z \rightarrow\ G := G \cup \{U\} \\
\qquad\quad [\!]\ U \in G \cup Z \rightarrow\ \textbf{skip} \\
\qquad\quad \textbf{fi} \\
\qquad ;\ \delta(T,a) := U \\
\quad \textbf{od} \\
\textbf{od} \\
;\ D := Z\ ;\ s := \{\mathit{First}(e' \cdot @)\}\ ;\ \mathit{Ac} := \{U \in D \mid @ \in U\} \\
\textbf{where } Q = S(e') \cup \{@\}
\end{array}
$$

□

This strongly resembles algorithm 3.5 in section 3.9 of [ASU]. The only essential difference is the treatment of the empty set as state of the deterministic automaton. In [ASU] the empty set is excluded from the states of the constructed deterministic automaton, although the transition function can have the empty set as an entry. In our opinion that is incorrect, either the empty set is treated as a ordinary state (sometimes called the "dead state"), or it is totally excluded (in that case a partial automaton is obtained). Here we have chosen the first version. The marking used in [ASU] is the second one described in Section 5. The functions *firstpos*, *lastpos* and *nullable*, used in [ASU], correspond to our functions *First*, *Last* and *Null*. Furthermore the function *followpos* is related to our relation[8] *Follow* by: $q \in \mathit{followpos}(p) = (p,q) \in \mathit{Follow}(e' \cdot @)$.

7 Conclusions

We have given a simple correctness proof of the automata constructions of McNaughton-Yamada [MY], Glushkov [Glu] and Berry-Sethi [BS] and a related method described by Aho, Sethi and Ullman [ASU]. The correctness proof of both constructions relies on theorem 19, i.e. $C(e) \Rightarrow \mathcal{K}(e) = \mathcal{L}(e)$. The only difference between the proofs for the two methods consists of the proof of $\mathbb{L}(\mathcal{M}_G(e)) = \mathcal{K}(e)$ for the Glushkov McNaughton-Yamada Berry-Sethi version versus $\mathbb{L}(\mathcal{M}_A(e)) = \mathcal{K}(e)$ for the Aho-Sethi-Ullman version. Moreover, the construction is given in two steps: (i) construct the intermediate nondeterministic automaton ($\mathit{unmark}_4(\mathcal{M}_G(e'))$ resp. $\mathit{unmark}_4(\mathcal{M}_A(e'))$) and (ii) apply the subset-reachable construction to it. Since step (ii) is a standard construction, this greatly simplifies the correctness proofs. Only in [BS] are these two steps also separated. Most computations are done in terms of regular expressions and languages. Algebras of automata, as in [Wa], or derivatives of regular expressions, as in [BS], are not used.
Finally we remark that the condition $C(e)$ in theorem 19 is too strong. For instance for $e = (a \mid a \cdot b)^*$ the condition does not hold, but still $\mathcal{K}(e) = \mathcal{L}(e)$.

References

[ASU] Aho, A.V., R.Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Reading, MA, 1988.

[Br] Brzozowski, J.A. "Derivatives of regular expressions", J.ACM 11(4):481-494, 1964.

[BS] Berry, G. and R. Sethi. "From regular expressions to deterministic automata", Theoretical Computer Science, 48: 117-126, 1986.

[Glu] Glushkov, V.M. "The abstract theory of automata", Russian Mathematical Surveys, 16: 1-53, 1961

[MY] McNaughton, R. and H. Yamada. "Regular expressions and state graphs for automata",

[8] The regular expression under consideration is not explicitly mentioned in [ASU].

IEEE Trans. on Electronic Computers 9(1):39-47, 1960.

[Wa] Watson, B.W. "A taxonomy of finite automata transformations and constructions", internal report, Eindhoven University of Technology, 1993.

In this series appeared:

91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.
91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.

91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.
91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.
91/35	F.S. de Boer J.W. Klop	Asynchronous communication in process algebra, p. 20.

	C. Palamidessi	
92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.
92/04	J.P.H.W.v.d.Eijnde	Program derivation in acyclic graphs and related problems, p. 90.
92/05	J.P.H.W.v.d.Eijnde	Conservative fixpoint functions on a graph, p. 25.
92/06	J.C.M. Baeten J.A. Bergstra	Discrete time process algebra, p.45.
92/07	R.P. Nederpelt	The fine-structure of lambda calculus, p. 110.
92/08	R.P. Nederpelt F. Kamareddine	On stepwise explicit substitution, p. 30.
92/09	R.C. Backhouse	Calculating the Warshall/Floyd path algorithm, p. 14.
92/10	P.M.P. Rambags	Composition and decomposition in a CPN model, p. 55.
92/11	R.C. Backhouse J.S.C.P.v.d.Woude	Demonic operators and monotype factors, p. 29.
92/12	F. Kamareddine	Set theory and nominalisation, Part I, p.26.
92/13	F. Kamareddine	Set theory and nominalisation, Part II, p.22.
92/14	J.C.M. Baeten	The total order assumption, p. 10.
92/15	F. Kamareddine	A system at the cross-roads of functional and logic programming, p.36.
92/16	R.R. Seljée	Integrity checking in deductive databases; an exposition, p.32.
92/17	W.M.P. van der Aalst	Interval timed coloured Petri nets and their analysis, p. 20.
92/18	R.Nederpelt F. Kamareddine	A unified approach to Type Theory through a refined lambda-calculus, p. 30.
92/19	J.C.M.Baeten J.A.Bergstra S.A.Smolka	Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.
92/20	F.Kamareddine	Are Types for Natural Language? P. 32.
92/21	F.Kamareddine	Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.

92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottum-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for Fω, p. 15.
92/26	T.H.W.Beelen W.J.J.Stut P.A.C.Verkoulen	A modelling method using MOVIE and SimCon/ExSpect, p. 15.
92/27	B. Watson G. Zwaan	A taxonomy of keyword pattern matching algorithms, p. 50.
93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.

92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottum-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for Fω, p. 15.
92/26	T.H.W.Beelen W.J.J.Stut P.A.C.Verkoulen	A modelling method using MOVIE and SimCon/ExSpect, p. 15.
92/27	B. Watson G. Zwaan	A taxonomy of keyword pattern matching algorithms, p. 50.
93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.
93/14	J.C.M. Baeten J.A. Bergstra	On Sequential Composition, Action Prefixes and Process Prefix, p. 21.

93/15	J.C.M. Baeten J.A. Bergstra R.N. Bol	A Real-Time Process Logic, p. 31.
93/16	H. Schepers J. Hooman	A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
93/17	D. Alstein P. van der Stok	Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
93/18	C. Verhoef	A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
93/19	G-J. Houben	The Design of an Online Help Facility for ExSpect, p.21.
93/20	F.S. de Boer	A Process Algebra of Concurrent Constraint Programming, p. 15.
93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correctness?, p. 24.
93/22	E. Poll	A Typechecker for Bijective Pure Type Systems, p. 28.
93/23	E. de Kogel	Relational Algebra and Equational Proofs, p. 23.
93/24	E. Poll and Paula Severi	Pure Type Systems with Definitions.
93/25	H. Schepers and R. Gerth	A Compositional Proof Theory for Fault Tolerant Real-Time Distributed Systems, p. 31.
93/26	W.M.P. van der Aalst	Multi-dimensional Petri nets, p. 25.
93/27	T. Kloks and D. Kratsch	Finding all minimal separators of a graph, p. 11.
93/28	F. Kamareddine and R. Nederpelt	A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.
93/29	R. Post and P. De Bra	GOLD, a Graph Oriented Language for Databases, p. 42.
93/30	J. Deogun T. Kloks D. Kratsch H. Müller	On Vertex Ranking for Permutation and Other Graphs, p. 11.
93/31	W. Körver	Derivation of Delay Insensitive and Speed Independent CMOS Circuits, using Directed Commands and Production Rule Sets, p. 39.