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

Citation for published version (APA):

Document status and date:
Published: 01/01/1993

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
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
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

All rights reserved
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 1
2 Preliminaries 1
3 A basic property 3
4 Automata for a restricted class of regular expressions 6
5 Automata for arbitrary regular expressions 10
6 Deterministic automata 13
7 Conclusions 16
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 \(L(e)\) corresponding to a given regular expression \(e\), will be defined. In Section 3, an alternative "language" \(K(e)\) corresponding to \(e\) is defined. In general \(L(e) \subseteq K(e)\). 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 \(K(e)\). Hence, if the condition \(C(e)\) holds, we have automata accepting \(L(e)\). 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 \(L(e')\) can easily be adapted such that they accept \(L(e)\). 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** \((RE_v, \text{ regular expressions over } V)\) The set \(RE_v\) of regular expressions over \(V\) is the smallest set \(X\) satisfying the following rules. For \(a \in V, e, f \in X:\)

\[
\begin{align*}
\epsilon & \in X \\
a & \in X \\
(e \lor f) & \in X \\
(e \cdot f) & \in X \\
e'^* & \in X
\end{align*}
\]

As usual we shall reduce the number of parentheses by: (i) giving priorities to the operators: \(\text{prio}(*)) > \text{prio}() > \text{prio}()\), (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 (Language defined by regular expression) The mapping $L_v : RE_v \rightarrow \mathcal{P}(V^*)$ is defined recursively as follows. For all $a \in V, e, f \in RE_v$:

\[
\begin{align*}
L_v(a) &= \{a\} \\
L_v(e \cdot f) &= L_v(e) \cup L_v(f) \\
L_v(e^*) &= (L_v(e))^* \\
\end{align*}
\]

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

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

\[
\begin{align*}
Null(e) &= \text{true} \\
Null(a) &= \text{false} \\
Null(e \cdot f) &= Null(e) \vee Null(f) \\
Null(e \cdot f) &= Null(e) \land Null(f) \\
Null(e^*) &= \text{true} \\
\end{align*}
\]

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

\[
\begin{align*}
First(e) &= \emptyset \\
First(a) &= \{a\} \\
First(e \cdot f) &= First(e) \cup First(f) \\
First(e \cdot f) &= \begin{cases} 
First(e) & \text{if } \neg Null(e) \\
First(e) \cup First(f) & \text{otherwise}
\end{cases} \\
First(e^*) &= First(e) \\
\end{align*}
\]

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

\[
\begin{align*}
Last(e) &= \emptyset \\
Last(a) &= \{a\} \\
Last(e \cdot f) &= Last(e) \cup Last(f) \\
Last(e \cdot f) &= \begin{cases} 
Last(f) & \text{if } \neg Null(f) \\
Last(e) \cup Last(f) & \text{otherwise}
\end{cases} \\
Last(e^*) &= Last(e) \\
\end{align*}
\]

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

\[
\begin{align*}
Follow(e) &= \emptyset \\
Follow(a) &= \emptyset \\
\end{align*}
\]
\[ \text{Follow}(e \mid f) = \text{Follow}(e) \cup \text{Follow}(f) \]
\[ \text{Follow}(e \cdot f) = \text{Follow}(e) \cup \text{Follow}(f) \cup (\text{Last}(e) \times \text{First}(f)) \]
\[ \text{Follow}(e^*) = \text{Follow}(e) \cup (\text{Last}(e) \times \text{First}(e)) \]

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

\[
\begin{align*}
\text{(7)} & \quad w \in \mathcal{L}(e) \\
& \Rightarrow \\
& (w = \varepsilon \land \text{Null}(e)) \lor \\
& (w_1 \in \text{First}(e) \land w_n \in \text{Last}(e) \land (\forall i : 1 \leq i < n : (w_i, w_{i+1}) \in \text{Follow}(e)))
\end{align*}
\]

where \( n = |w| \). Note that in general the reverse implication does not hold, take for instance \( e = \varepsilon a \) and \( w = \varepsilon aa \). 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 ( \( \text{in}, \# \))** The mapping \( \text{in} : V \times RE_V \to \mathbb{B} \) (written in infix notation) is defined recursively as follows. For all \( a, b \in V, e, f \in RE_V \):

- \( a \text{ in } e = \text{false} \)
- \( a \text{ in } e \cdot f = a \text{ in } e \lor a \text{ in } f \)
- \( a \text{ in } e^* = a \text{ in } e \)

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

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

So \( a \text{ in } e \) means that the alphabet symbol \( a \) occurs somewhere in the regular expression \( e \) and \( \#(a, e) \) denotes the number of occurrences of \( a \) in \( e \).
Definition 9 \((C, S)\) The mapping \(C : \mathcal{R} \to \mathcal{I} \mathcal{B}\) is defined as follows. For all \(e \in \mathcal{R}\):

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

The mapping \(S : \mathcal{R} \to \mathcal{P}(V)\) is defined as follows. For all \(e \in \mathcal{R}\):

\[
S(e) = \{a \in V \mid a \text{ in } e\}
\]

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.

\[
\begin{align*}
(10) & \quad C(e_1 | e_2) = C(e_1) \land C(e_2) \land (S(e_1) \cap S(e_2) = \emptyset) \\
(11) & \quad C(e_1 \cdot e_2) = C(e_1) \land C(e_2) \land (S(e_1) \cap S(e_2) = \emptyset) \\
(12) & \quad C(e^*) = C(e)
\end{align*}
\]

To formulate the main theorem of this section we give the following definition\(^2\).

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

\[
P(w, e) =
\begin{align*}
& (w = e \land \text{Null}(e)) \lor \\
& (w_1 \in \text{First}(e) \land w_n \in \text{Last}(e) \land (\forall i : 1 \leq i < n : (w_i, w_{i+1}) \in \text{Follow}(e))
\end{align*}
\]

where \(n = |w|\).

The mapping \(\mathcal{K} : \mathcal{R} \to \mathcal{P}(V^*)\) 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 \(\mathcal{K}(e) = \{e\}\)

\(^2\)Note that \(P(w, e)\) is equal to the right hand side of (7).
Proof: Trivially $P(w, \varepsilon) = (w = \varepsilon)$.
\[\square\]

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

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

Lemma 16 For all $e_1, e_2 \in RE_V$ with $S(e_1) \cap S(e_2) = \emptyset$:
\[K(e_1 | e_2) = K(e_1) \cup K(e_2)\]

Proof: It is sufficient to prove that for all $w \in V^*$:
\[P(w, e_1 | e_2) = P(w, e_1) \lor P(w, e_2)\]
\[\iff \text{Trivially for all } e_1, e_2 \in RE_V : P(w, e_1) \lor P(w, e_2) \Rightarrow P(w, e_1 | e_2)\]
\[\Rightarrow \text{For } w = \varepsilon \text{ this follows immediately from the definition of } \text{Null}. \text{ Using } S(e_1) \cap S(e_2) = \emptyset \text{ we observe that for } w \neq \varepsilon : P(w, e_1 | e_2) \land w_1 \in S(e_i) \Rightarrow w \in S(e_i)^* \text{ for } i = 1, 2. \text{ Hence, for } w \neq \varepsilon : P(w, e_1 | e_2) \land w_1 \in S(e_i) \Rightarrow P(w, e_i) \text{ for } i = 1, 2.\]
\[\square\]

Lemma 17 For all $e_1, e_2 \in RE_V$ with $S(e_1) \cap S(e_2) = \emptyset$:
\[K(e_1 \cdot e_2) = K(e_1) \cdot K(e_2)\]

Proof: It is sufficient to prove that for all $w \in V^*$:
\[P(w, e_1 \cdot e_2) = (\exists u, v \in V^* : w = uv : P(u, e_1) \land P(v, e_2))\]
\[\iff \text{Trivially for all } e_1, e_2 \in RE_V : P(u, e_1) \land P(v, e_2) \Rightarrow P(uv, e_1 \cdot e_2)\]
\[\Rightarrow \text{This implication is proved by using the observation that, since } S(e_1) \cap S(e_2) = \emptyset, \text{ we have } P(w, e_1 \cdot e_2) \Rightarrow w \in S(e_1)^* S(e_2)^*.\]
\[\square\]

Lemma 18 For all $e \in RE_V$:
\[K(e^*) = K(e)\]

Proof: It is sufficient to prove that for all $w \in V^*$:
\[P(w, e^*) = w \in K(e)\]
\[\iff \text{Trivially } P(e, e^*) \text{ and } P(u, e) \land P(v, e^*) \Rightarrow P(uv, e^*).\]
\[\Rightarrow \text{Assume } P(w, e^*) \text{ holds. Using } \text{Follow}(e^*) = \text{Follow}(e) \cup (\text{Last}(e) \times \text{First}(e)) \text{ we conclude that there exists a (possibly empty) index set } I \subseteq \{1, \ldots, |w| - 1\} \text{ such that}\]
\[
\text{(i)} : i \leq |w| \land i \notin I : (w_i, w_{i+1}) \in \text{Follow}(e)) \land \\
\text{(ii)} : i \in I : (w_i, w_{i+1}) \in (\text{Last}(e) \times \text{First}(e)) \setminus \text{Follow}(e))
\]
Hence $w$ can be written as a concatenation of $I = |I| + 1$ strings, say $w = v^1 \ldots v^l$ such that $P(v^i, e)$ holds for all $i \in \{1, \ldots , I\}$. This means that $w \in \mathcal{K}(e)^I \subseteq \mathcal{K}(e)^*$. 

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. 

\[ \square \]

### 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 $(Q, W, \Delta, S, F)$ with:

- $Q$ a finite set of states,
- $W$ a finite alphabet,
- $\Delta$, 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.

\[ \square \]

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

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

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

\[ \square \]

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 $\mathcal{M}_G : RE_V \rightarrow SNFA_v$ is defined by $\mathcal{M}_G(e) = (Q, V, \Delta, \{\varepsilon\}, F)$ where:

- $\varepsilon \notin V$
- $Q = S(e) \cup \{\varepsilon\}$
- $F = \text{Last}(\varepsilon \cdot e)$
\( \Delta \subseteq Q \times V \times Q \text{ is such that} \)
\[ (p, a, q) \in \Delta = (p, q) \in \text{Follow}(\@ \cdot e) \land a = q \]

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 \( \mathcal{M}_C(e) \) is always a deterministic automaton because \( (p, a, q_1) \in \Delta \land (p, a, q_2) \in \Delta \Rightarrow q_1 = q_2 \).

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

- \( \@ \not\in V \)
- \( Q = S(e) \cup \{\@\} \)
- \( S = \text{First}(e \cdot \@) \)
- \( \Delta \subseteq Q \times V \times Q \text{ is such that} \)
\[ (p, a, q) \in \Delta = (p, q) \in \text{Follow}(e \cdot \@) \land a = p \]

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 \( e \in REV \):
\[ L(\mathcal{M}_C(e)) = K(e) \]

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

\[ w \in L(M) = \{ \text{definition 21} \}
\[ (\exists q_0, \ldots, q_n \in Q :: q_0 \in S \land q_n \in F \land (\forall i : 0 \leq i < n :: (q_i, w_{i+1}, q_{i+1}) \in \Delta)) \]
\[ = \{ \text{definition 22} \}
\[ (\exists q_0, \ldots, q_n \in Q :: q_0 = @ \land q_n \in \text{Last}(\@ \cdot e) \land
\[ (\forall i : 0 \leq i < n :: w_{i+1} = q_{i+1} \land (q_i, q_{i+1}) \in (\{\@\} \times \text{First}(e)) \cup \text{Follow}(e)) \]
\[ = \{ n > 0 \}
\[ (@, w_1) \in (\{\@\} \times \text{First}(e)) \cup \text{Follow}(e) \land w_n \in \text{Last}(\@ \cdot e) \land
\[ (\forall i : 0 < i < n :: (w_i, w_{i+1}) \in (\{\@\} \times \text{First}(e)) \cup \text{Follow}(e)) \]
\[ = \{ @ \not\in V \}
\[ w_1 \in \text{First}(e) \land w_n \in \text{Last}(e) \land
\[ (\forall i : 1 \leq i < n :: (w_i, w_{i+1}) \in \text{Follow}(e)) \]
\[ = \{ \text{definition 13} \}
\[ w \in K(e) \]

For the empty string:
\[ e \in L(M) = \{ \text{definition } \mathcal{M} \}
\[ @ \in \text{Last}(\@ \cdot e) \]
\[ = \{ @ \not\in V \}

3Note that the reverse automaton, i.e. the automaton obtained by reversing all transition arrows and interchanging initial and final states, is deterministic.
\[ \text{Null}(e) = \{ \text{definition13} \} \]

\[ e \in \mathcal{K}(e) \]

\[ \square \]

**Theorem 25** For all \( e \in RE_v \):

\[ \mathcal{L}(\mathcal{M}_A(e)) = \mathcal{K}(e) \]

**Proof:** Similarly to the proof of theorem 24.

\[ \square \]

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

\[ \mathcal{L}(\mathcal{M}_G(e)) = \mathcal{L}(e) \]

\[ \mathcal{L}(\mathcal{M}_A(e)) = \mathcal{L}(e) \]

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

\[ \square \]

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

\[ S(e) = \{1, 2, 3\} \]

\[ \text{Last}(\emptyset \cdot e) = \{1, 3\} \]

\[ \text{Follow}(\emptyset \cdot e) = \{\{\emptyset, 1\}, \{\emptyset, 2\}, \{\emptyset, 3\}, \{2, 2\}, \{2, 3\}\} \]

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

\[ Q = S(e) \cup \{\emptyset\} = \{1, 2, 3, \emptyset\} \]

\[ F = \text{Last}(\emptyset \cdot e) = \{1, 3\} \]

\[ \Delta = \{(p, q, q) \mid (p, q) \in \text{Follow}(\emptyset \cdot e)\} = \{\{\emptyset, 1, 1\}, \{\emptyset, 2, 2\}, \{\emptyset, 3, 3\}, \{2, 2, 2\}, \{2, 3, 3\}\} \]
The automaton has the following graphical representation:\(^4\).

Note that this is indeed a deterministic automaton. Furthermore

\[
\text{First}(e \cdot \mathord{\varepsilon}) = \{1, 2, 3\}
\]
\[
\text{Follow}(e \cdot \mathord{\varepsilon}) = \{(1, \mathord{\varepsilon}), (2, 2), (2, 3), (3, \mathord{\varepsilon})\}
\]

Hence the automaton \(M_A(e) = (Q, V, \Delta, S, \{\mathord{\varepsilon}\})\) is given by

\[
Q = S(e) \cup \{\mathord{\varepsilon}\} = \{1, 2, 3, \mathord{\varepsilon}\}
\]
\[
S = \text{First}(e \cdot \mathord{\varepsilon}) = \{1, 2, 3\}
\]
\[
\Delta = \{(p, p, q) | (p, q \in \text{Follow}(e \cdot \mathord{\varepsilon}))\} = \{(1, 1, \mathord{\varepsilon}), (2, 2, 2), (2, 2, 3), (3, 3, \mathord{\varepsilon})\}
\]

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 \(L(e)\).

\(\square\)

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

\[Rev_M : SNFA \rightarrow SNFA\] by

\(^4\)Start states will be depicted with an unlabeled incoming arrow and final states will be depicted by two concentric circles.
\[ \text{Rev}_M((Q, V, \Delta, S, F)) = (Q, V, \Delta', F, S) \]

with \((p, a, q) \in \Delta = (q, a, p) \in \Delta'\)

and

\[
\begin{align*}
\text{Rev}_e : RE_V & \rightarrow RE_V \text{ by} \\
\text{Rev}_e(e) & = e \\
\text{Rev}_e(a) & = a \text{ for all } a \in V \\
\text{Rev}_e(e_1 | e_2) & = \text{Rev}_e(e_2) | \text{Rev}_e(e_1) \\
\text{Rev}_e(e_1 \cdot e_2) & = \text{Rev}_e(e_2) \cdot \text{Rev}_e(e_1) \\
\text{Rev}_e(e^*) & = \text{Rev}_e(e)^* \\
\end{align*}
\]

So \(\text{Rev}_M\) reverses SNFA's and \(\text{Rev}_e\) reverses regular expressions. Then, for all \(e\), the following relations hold:

\[
\begin{align*}
\text{Rev}_M(M_G(e)) &= M_A(\text{Rev}_e(e)) \\
\text{Rev}_M(M_A(e)) &= M_G(\text{Rev}_e(e)) \\
\end{align*}
\]

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 \(L_V(e')\) can easily be adapted to an automaton accepting the original language \(L_V(e)\).

Definition 28 (Marking) A marking with respect to a regular expression \(e\) over an alphabet \(V\) is a triple \(<V', e', \text{unmark}>\) such that:
- \(V'\) is an alphabet
- \(e' \in RE_V\)
- \(C(e')\) holds
- \(\text{unmark} : V' \rightarrow V\) is such that \(e = \text{unmark}_1(e')\)

where \(\text{unmark}_1 : RE_V \rightarrow RE_V\) is defined as the unique homomorphic extension\(^5\) of \(\text{unmark}\) to \(RE_V\).

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

Definition 29 (Unmarkings) The mappings \(\text{unmark}_2 : V' \rightarrow V^*\), \(\text{unmark}_3 : \mathcal{P}(V') \rightarrow \mathcal{P}(V^*)\) and \(\text{unmark}_4 : SNFA_V \rightarrow SNFA_V\) are defined as follows:

\[
\begin{align*}
\text{unmark}_2(e) & = e \\
\text{unmark}_2(a) & = \text{unmark}(a) \text{ for all } a \in V' \\
\end{align*}
\]

\[\text{i.e. } \text{unmark}_2(e) = e, \text{unmark}_3(a) = \text{unmark}(a), \text{unmark}_4(e_1 | e_2) = \text{unmark}_4(e_2) | \text{unmark}_4(e_1), \text{unmark}_4(e_1 \cdot e_2) = \text{unmark}_4(e_1) \cdot \text{unmark}_4(e_2), \text{and unmark}_4(f^*) = (\text{unmark}_4(f))^*\]
unmark₂(xy) = unmark₂(x)unmark₂(y)  for all x, y ∈ V''
unmark₃(L) = {unmark₂(x) | x ∈ L}
unmark₄((Q, V', Δ', S, F)) = (Q, V, Δ, S, F) with
Δ = {(p, unmark(a), q) | (p, a, q) ∈ Δ'}

So the mapping unmark : V' → V gives rise to similar "unmarkings" for regular expressions, strings, languages and finite automata.

Lemma 30 unmark₃(L) = L ∘ unmark₁

Proof: It is easily shown using induction that for all e'' ∈ RE_v : unmark₃(L_v(e'')) = L_v(unmark₁(e'')). □

Lemma 31 unmark₄(L_v) = L ∘ unmark₄

Proof: Let M = (Q, V', Δ, S, F) be an SNFA over V'. Then unmark₄(M) = (Q, V, Δ, S, F) with Δ = {(p, unmark(a), q) | (p, a, q) ∈ Δ'}. We have to show that unmark₃(L_v(M)) = L_v(unmark₄(M)).

Let w ∈ V* with n = |w|. Then

w ∈ L_v(unmark₄(M))
= { definition 21 }
(∃q₀,...,qₙ ∈ Q : q₀ ∈ S ∧ qₙ ∈ F ∧ (∀i : 0 ≤ i < n : (qᵢ, wᵢ₊₁, qᵢ₊₁) ∈ Δ))
= { (p, a, q) ∈ Δ = (∃a' ∈ V' : a = unmark(a') : (p, a', q) ∈ Δ') }
(∃q₀,...,qₙ ∈ Q : q₀ ∈ S ∧ qₙ ∈ F ∧ (∃a' ∈ V' : wᵢ₊₁ = unmark(a') : (qᵢ, a', qᵢ₊₁) ∈ Δ'))
= { interchanging quantifications }
(∃q₀,...,qₙ ∈ Q : q₀ ∈ S ∧ qₙ ∈ F ∧ (∃v ∈ V* : w = unmark₂(v) : (∀i : 0 ≤ i < n : (qᵢ, vᵢ₊₁, qᵢ₊₁) ∈ Δ'))
= { interchanging quantifications }
(∃v ∈ V* : w = unmark₂(v) :
(∃q₀,...,qₙ ∈ Q : q₀ ∈ S ∧ qₙ ∈ F ∧ (∀i : 0 ≤ i < n : (qᵢ, vᵢ₊₁, qᵢ₊₁) ∈ Δ'))
= { definition 21 }
(∃v ∈ V* : w = unmark₂(v) : v ∈ L_v(M))
= w ∈ unmark₃(L_v(M))

□

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 L(e).

---

6The sets RE_v, P(V) and SNFA can all be given a S-algebra structure, see for instance [Wa]. Then various mappings can be considered as S-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 unmark₄(Mₑ') and unmark₄(Mₐₑ') accept the language L(e).

Proof:

\[ \mathcal{L}_V(\text{unmark}_4(Mₐₑ')) \]
\[ = \{ \text{lemma 31} \} \]
\[ \text{unmark}_3(\mathcal{L}_V(Mₐₑ')) \]
\[ = \{ C(e'), \text{ theorem 26} \} \]
\[ \text{unmark}_3(\mathcal{L}_V(e')) \]
\[ = \{ \text{lemma 30} \} \]
\[ \mathcal{L}_V(\text{unmark}_3(e')) \]
\[ = \{ \text{unmark}_1(e') = e \} \]
\[ \mathcal{L}_V(e) \]

The proof for the automaton unmark₄(Mₐₑ') proceeds similarly.

So far we have assumed that \( < V', e', \text{unmark} > \) 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 \( < a, \text{position of this occurrence of } a \text{ in } e > \). Then \( V' = V \times \{1, ..., n\} \) where \( n \) is the number of occurrences of alphabet symbols in \( e \), and the mapping \( \text{unmark} : V' \rightarrow V \) is projection on the first coordinate.
  
  For instance the expression \( e = a \cdot (a \cdot b) \cdot b \) over \( V = \{a, b\} \) is mapped onto \( e' = < a, 1 > \cdot (< a, 2 > < b, 3 >) \cdot < b, 4 > \) over \( V' = V \times \{1, ..., 4\} \). 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 \( V' = \{1, ..., n\} \) where \( n \) is the number of occurrences of alphabet symbols and the mapping \( \text{unmark} : V' \rightarrow V \) is defined by
  
  \[ \text{unmark}(k) = "\text{symbol at position } k \text{ in } e" \]

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

Example
Consider the regular expression \( e = a \cdot a \cdot b \). As the symbol \( a \) appears twice, it does not satisfy the condition \( C(e) \). Following the second method above \( < \{1, 2, 3\}, e', \text{unmark} > \) with \( e' = 1 \cdot 2 \cdot 3 \) and \( \text{unmark}(1) = a, \text{unmark}(2) = a, \text{unmark}(3) = b \) is a marking of \( e \). Note that \( e' \) equals the regular expression used in example 27. Hence the automata \( Mₐₑ' \) and \( Mₐₑ' \) are equal to the automata given in example 27. After unmarking we obtain the automaton \( \text{unmark}_4(Mₐₑ') \) with graphical representation.
Theorem 32 implies that both automata accept the language $L(e)$.

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 $M_G$ or $M_A$, as described in section 4, to $e'$, thus obtaining an SNFA accepting $L(e')$,
(c) unmark the obtained SNFA to obtain an SNFA accepting $L(e)$.

The intermediate automaton $M_G(e)$ is deterministic, while, in general, the intermediate automaton $M_A(e)$ is not deterministic. However, due to the unmarking process we will generally end up with a non-deterministic automaton. Since we ultimately want a deterministic automaton accepting $L(e)$, 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

\footnote{If unmark is not injective, the mapping unmark\textsubscript{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 $(Q, V, \delta, s, A')$ with:

- $Q$ a finite set of states,
- $V$ a finite alphabet,
- $\delta$, the transition mapping, is a mapping $Q \times V \rightarrow Q$,
- $s \in Q$ is the start state,
- $A' \subseteq Q$ is the set of final states.

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

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

$\forall w \in V^*$

$L'(M) = \{ w \in V^* | \delta^*(s, w) \in F \}$

where the mapping $\delta^* : Q \times V^* \rightarrow Q$ is defined by:

$\delta^*(q, e) = q$

$\delta^*(q, aw) = \delta^*(\delta(q, a), w)$

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

$\text{Subset}((Q, V, \Delta, s, F)) = (\mathcal{P}(Q), V, \delta, \{s\}, F_1)$

where

$\delta(R, a) = \{ q \in Q | (\exists r \in R : (r, a, q) \in \Delta) \}$

$F_1 = \{ R \in \mathcal{P}(Q) | R \cap F \neq \emptyset \}$

Due to the absence of $e$-transitions the subset construction for $\mathcal{SNFA}$'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) $L' \circ \text{Subset} = L$

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

$\text{Reachable}((Q, V, \delta, s, F)) = (Q_1, V, \delta_1, s, F_1)$

with

$Q_1 = \{ \delta^*(s, w) | w \in V^* \}$

$\delta_1 = \delta \upharpoonright_{Q_1 \times V}$

$F_1 = F \cap Q_1$

where $\delta^*$ is the mapping introduced in definition 34

Trivially the elimination of unreachable states does not change the language accepted by the automaton, i.e.
The composition \( \text{Reachable}_o \) \( \text{Subset} \) corresponds to the subset-reachable construction mentioned above. From (36) and (38) we obtain that

\[
(39) \quad \text{IL'} \circ \text{Reachable}_o \text{Subset} = \text{IL}
\]

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

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

\[
\text{Reachable}_o \text{Subset}((Q, V, \Delta, S, F)) = (D, V, \delta, s, Ac)
\]

where \( D, \delta, s, Ac \) 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
  - \( \delta(T, a) := U \)
  - 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 \( <V', e', \text{unmark}> \) 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):

\[
\text{unmark}_A(M_A(e')) = (Q, V, \Delta, S, \{\emptyset\})
\]

where

- \( Q = S(e') \cup \{\emptyset\} \)
- \( \Delta = \{(p, \text{unmark}(a), q) \mid (p, q) \in \text{Follow}(e' \cdot \emptyset) \land a = p\} \)
- \( S = \text{First}(e' \cdot \emptyset) \)

Applying the subset-reachable construction to this SNFA yields:

\[
\text{Subset \ Reachable}(\text{unmark}_A(M_A(e'))) = (D, V, \delta, s, Ac)
\]

where \( D, \delta, s, Ac \) are computed by the following algorithm:

- var \( Z, G : \mathcal{P}(\mathcal{P}(Q)) ; U, T : \mathcal{P}(Q) ; a : V \)
- \( | Z := \emptyset ; G := \{\text{First}(e' \cdot \emptyset)\} \)
- do \( G \neq \emptyset \rightarrow \)
  - let \( T \in G \)
\[ Z := Z \cup \{T\} \;
G := G \setminus \{T\} \]

\begin{verbatim}
for all \( a \in V \) do
  \[ U := \{ q \in Q \mid (\exists p \in T \colon (p, q) \in \text{Follow}(e' \cdot @) \land a = \text{unmark}(p)) \} \]
  if \( U \notin G \cup Z \rightarrow G := G \cup \{U\} \]
  \[ \delta(T, a) := U \]
od
\end{verbatim}

\[ D := Z ; \; s := \{ \text{First}(e' \cdot @) \} ; \; Ac := \{ U \in D \mid \emptyset \in U \} \]

where \( Q = S(e') \cup \{\emptyset\} \)

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 \text{firstpos}, \text{lastpos} and \text{nullable}, used in [ASU], correspond to our functions \text{First}, \text{Last} and \text{Null}. Furthermore the function \text{followpos} is related to our relation 8 \text{Follow} by: \( q \in \text{followpos}(p) = (p, q) \in \text{Follow}(e' \cdot @) \).

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 \( IL(MG(e)) = \mathcal{K}(e) \) for the Glushkov McNaughton-Yamada Berry-Sethi version versus \( IL(MA(e)) = \mathcal{K}(e) \) for the Aho-Sethi-Ullman version. Moreover, the construction is given in two steps: (i) construct the intermediate nondeterministic automaton \((\text{unmark}_4(M_G(e'))}\) resp. \((\text{unmark}_4(M_A(e'))}\) 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


[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].

In this series appeared:

91/01 D. Alstein


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


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


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.
<table>
<thead>
<tr>
<th>Page</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>91/18</td>
<td>Rik van Geldrop</td>
<td>Transformational Query Solving, p. 35.</td>
</tr>
<tr>
<td>91/19</td>
<td>Erik Poll</td>
<td>Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.</td>
</tr>
<tr>
<td>91/23</td>
<td>K.M. van Hee, L.J. Somers, M. Voorhoeve</td>
<td>Z and high level Petri nets, p. 16.</td>
</tr>
<tr>
<td>91/24</td>
<td>A.T.M. Aerts, D. de Reus</td>
<td>Formal semantics for BRM with examples, p. 25.</td>
</tr>
<tr>
<td>91/25</td>
<td>P. Zhou, J. Hooman, R. Kuiper</td>
<td>A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and completeness, p. 52.</td>
</tr>
<tr>
<td>91/27</td>
<td>F. de Boer, C. Palamidessi</td>
<td>Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.</td>
</tr>
<tr>
<td>91/28</td>
<td>F. de Boer</td>
<td>A compositional proof system for dynamic process creation, p. 24.</td>
</tr>
<tr>
<td>91/30</td>
<td>J.C.M. Baeten, F.W. Vaandrager</td>
<td>An Algebra for Process Creation, p. 29.</td>
</tr>
<tr>
<td>91/31</td>
<td>H. ten Eikelder</td>
<td>Some algorithms to decide the equivalence of recursive types, p. 26.</td>
</tr>
<tr>
<td>91/33</td>
<td>W. v.d. Aalst</td>
<td>The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.</td>
</tr>
<tr>
<td>91/34</td>
<td>J. Coenen</td>
<td>Specifying fault tolerant programs in deontic logic, p. 15.</td>
</tr>
<tr>
<td>No.</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>-----</td>
<td>-------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>92/01</td>
<td>C. Palamidessi J. Coenen J. Zwiers W.-P. de Roever</td>
<td>A note on compositional refinement, p. 27.</td>
</tr>
<tr>
<td>92/02</td>
<td>J. Coenen J. Hooman</td>
<td>A compositional semantics for fault tolerant real-time systems, p. 18.</td>
</tr>
<tr>
<td>92/03</td>
<td>J.C.M. Baeten J.A. Bergstra</td>
<td>Real space process algebra, p. 42.</td>
</tr>
<tr>
<td>92/05</td>
<td>J.P.H.W.v.d.Eijnde</td>
<td>Conservative fixpoint functions on a graph, p. 25.</td>
</tr>
<tr>
<td>92/06</td>
<td>J.C.M. Baeten J.A. Bergstra</td>
<td>Discrete time process algebra, p.45.</td>
</tr>
<tr>
<td>92/07</td>
<td>R.P. Nederpelt</td>
<td>The fine-structure of lambda calculus, p. 110.</td>
</tr>
<tr>
<td>92/10</td>
<td>P.M.P. Rambags</td>
<td>Composition and decomposition in a CPN model, p. 55.</td>
</tr>
<tr>
<td>92/13</td>
<td>F. Kamareddine</td>
<td>Set theory and nominalisation, Part II, p.22.</td>
</tr>
<tr>
<td>92/14</td>
<td>J.C.M. Baeten</td>
<td>The total order assumption, p. 10.</td>
</tr>
<tr>
<td>92/15</td>
<td>F. Kamareddine</td>
<td>A system at the cross-roads of functional and logic programming, p.36.</td>
</tr>
<tr>
<td>92/16</td>
<td>R.R. Seljée</td>
<td>Integrity checking in deductive databases; an exposition, p.32.</td>
</tr>
<tr>
<td>92/17</td>
<td>W.M.P. van der Aalst</td>
<td>Interval timed coloured Petri nets and their analysis, p. 20.</td>
</tr>
<tr>
<td>92/18</td>
<td>R.Nederpelt F. Kamareddine</td>
<td>A unified approach to Type Theory through a refined lambda-calculus, p. 30.</td>
</tr>
<tr>
<td>92/19</td>
<td>J.C.M.Baeten J.A.Bergstra S.A.Smolka</td>
<td>Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.</td>
</tr>
<tr>
<td>92/20</td>
<td>F.Kamareddine</td>
<td>Are Types for Natural Language? P. 32.</td>
</tr>
<tr>
<td>92/21</td>
<td>F.Kamareddine</td>
<td>Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.</td>
</tr>
</tbody>
</table>
A useful lambda notation, p. 17.

Nominalization, Predication and Type Containment, p. 40.

Bottom-up Abstract Interpretation of Logic Programs, p. 33.

A Programming Logic for F0, p. 15.

A modelling method using MOVIE and SimCon/ExSpect, p. 15.

A taxonomy of keyword pattern matching algorithms, p. 50.

Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.

A continuous version of the Prisoner’s Dilemma, p. 17

Quicksort for linked lists, p. 8.

Deterministic and randomized local search, p. 78.

A congruence theorem for structured operational semantics with predicates, p. 18.

On the unavoidability of metastable behaviour, p. 29

Exercises in Multiprogramming, p. 97

A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.


Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.


<table>
<thead>
<tr>
<th>ISBN</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>92/22</td>
<td>R. Nederpelt, F. Kamareddine, E. Klein</td>
<td>A useful lambda notation, p. 17.</td>
</tr>
<tr>
<td>92/23</td>
<td>F. Kamareddine, E. Klein</td>
<td>Nominalization, Predication and Type Containment, p. 40.</td>
</tr>
<tr>
<td>92/24</td>
<td>M. Codish, D. Dams, Eyal Yardeni</td>
<td>Bottom-up Abstract Interpretation of Logic Programs, p. 33.</td>
</tr>
<tr>
<td>92/25</td>
<td>E. Poll</td>
<td>A Programming Logic for Foo, p. 15.</td>
</tr>
<tr>
<td>93/01</td>
<td>R. van Geldrop</td>
<td>Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.</td>
</tr>
<tr>
<td>93/02</td>
<td>T. Verhooff</td>
<td>A continuous version of the Prisoner's Dilemma, p. 17</td>
</tr>
<tr>
<td>93/03</td>
<td>T. Verhooff</td>
<td>Quicksort for linked lists, p. 8.</td>
</tr>
<tr>
<td>93/04</td>
<td>E. H. L. Aarts, J. H. M. Korst, P. J. Zwietering</td>
<td>Deterministic and randomized local search, p. 78.</td>
</tr>
<tr>
<td>93/05</td>
<td>J. C. M. Baeten, C. Verhoeff</td>
<td>A congruence theorem for structured operational semantics with predicates, p. 18.</td>
</tr>
<tr>
<td>93/06</td>
<td>J. P. Veltkamp</td>
<td>On the unavoidability of metastable behaviour, p. 29</td>
</tr>
<tr>
<td>93/07</td>
<td>P. D. Moerland</td>
<td>Exercises in Multiprogramming, p. 97</td>
</tr>
<tr>
<td>93/08</td>
<td>J. Verhoosel</td>
<td>A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.</td>
</tr>
<tr>
<td>93/10</td>
<td>K. M. van Hee</td>
<td>Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.</td>
</tr>
</tbody>
</table>
| 93/15 | J.C.M. Baeten  
|       | J.A. Bergstra  
| 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      
|       | G. Fité        |                                           |
|       | M. Bruynooghe  |                                           |
| 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. |
| 93/26 | 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       | On Vertex Ranking for Permutation and Other Graphs,  
|       | D. Kratsch    | p. 11. |
|       | H. Müller      |                                           |
| 93/31 | W. Körver      | Derivation of Delay Insensitive and Speed Independent  
|       |               | CMOS Circuits, using Directed Commands and  
|       |               | Production Rule Sets, p. 39. |