MASTER

FADELA : Finite Automata DEbugging LAnguage

van der Zanden, J.G.N.M.

Award date:
1990

tue

EINDHOVEN UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING
DESIGN AUTOMATION GROUP

FADELA
Finite Automata DEbugging LAnguage

J.G.N.M. van der Zanden

Master thesis
reporting on graduation work
performed from 09.1989 to 08.1990
by order of prof. dr. ing. J.A.G. Jess
and supervised by ir. G.G. de Jong.

ABSTRACT

Finite state models are a popular method used in checking the behaviour of complex systems, such as communication protocols.
These models can describe certain behaviours, that an implementation should have. If the implementation is also described by such a model, the theory of finite state models offers a well-described way of checking whether the implementation does match the desired behaviours.

In this report the theory of finite state automata is discussed, describing finite automata, the languages accepted by them and the regular expressions describing those languages. Methods to construct automata accepting a certain language, to construct a regular expression describing the language accepted by a given automaton and to construct a deterministic equivalent of a non-deterministic automaton are given.

An extension of the theory is given, which handles ω-regular expressions that describe languages consisting of words of infinite length. B- and M-automata accepting such languages are described. Again methods to derive expressions from given automata and automata from ω-regular expressions are given.

For M-automata, a method to directly convert a non-deterministic automaton into an equivalent deterministic one has been found. This method is based on the well-known subset-construction that is used to construct deterministic 'normal' automata.
After duplication of the limitsets of a non-deterministic M-automaton, subset-construction yields a deterministic version, in which the limitsets are easily recognized.

These theories are used to create an environment, where the user can define finite state models and perform several operations on them. The implementation of conversion between automata and regular expressions, simplification of regular expressions, complementation, determinization, minimization and several definitions of products of automata, lead to a flexible tool called FADELA, a Finite Automata DEbugging Language. The use of symbolic descriptions for states and inputs of automata, together with the possibility to include batch-files in interactive operation offer a powerful tool to examine finite state models and their equivalences.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

An important step in the design of complex systems is verification. One has to be sure, that the implementation of a system matches the specifications that were given for the system, i.e. the correctness of the implementation must be ensured.

In the case of communication protocols, when implementing such a protocol, one would for example like to prove the absence of deadlock or the existence of liveness. Informal specifications of protocols are often imprecise and may not be sufficient to ensure the correctness. Therefore, formal specification and verification techniques are being used.

One of the formal methods to describe communication protocols, that is very popular, is the use of finite-state models (finite automata). The (implementation of the) protocol is then described by an automaton. When the properties that this protocol should have, can also be described by automata, a formal verification is possible, based on the theory of finite state automata. The protocol is correct, with respect to the given properties, if and only if no accepted behaviour of the protocol-automaton is accepted by the automata that describe the negation of the desired properties, or, equivalently, if the set of accepted behaviors of the product-automaton of the automaton describing the protocol and the one describing the negation of the property is empty. [1,2]

To do this kind of verification, an environment is useful, in which the user can create his models in the form of automata or regular expressions describing such automata, where he can perform 'standard' operations on these items, describe more complex operations and view the results. Because often the infinite behaviour of a system is of interest, also automata accepting infinite sequences and their description by ω-regular expressions should be made accessible in this tool.

These considerations have lead to the development of FADELA, a Finite Automata DEbugging LAnguage, which enables the user to work interactively on automata and expressions, but also has the possibility of performing batch-jobs.

2. *FINITE AUTOMATA AND REGULAR EXPRESSIONS*

There is much literature on the theory of automata and languages or regular expressions. Some good handbooks and articles give definitions of these subjects [3,4,5,6]. However, to have a clear understanding of the denomination used in this report, in this chapter some definitions and notations will be given.

Descriptions of languages consisting of finite-length words, the regular expressions that describe these languages and the finite- state automata accepting them, are given.

2.1 Words, alphabets and languages

A language L is a set of words w.
The empty language, containing no words, is denoted by $\varnothing$.
Every word in a language is a sequence of symbols x. The finite set X of symbols that are used to form the words of the language, is called the alphabet over which the language is formed.
The length of w, denoted by $|w|$, is the number of terms (symbols) in w. For all i, $0 < i \leq |w|$, the i^{th} term in the word w is denoted by $w[i]$.
The one word that has length zero, is denoted by ε.
The (infinite) set of all finite words over X (including ε) is denoted by X^*.

Words can be concatenated as follows:

If u and v are words, having lengths $|u|$ and $|v|$ respectively, then $w = u.v$ is a word, for which

$$|w| = |u| + |v|$$
$$w[i] = \begin{cases} u[i] & 0 < i \leq |u| \\ v[i-|u|] & |u| < i \leq |w| \end{cases}$$

Given a word w, the repeated concatenation w^n is defined, for $n > 0$, as:

$$w^0 = \varepsilon$$
$$w^1 = w$$
$$w^n = w^{n-1}.w$$

Clearly it follows that $|w^n| = n\,|w|$.

The reverse w^R of a word w is defined as:

$$\varepsilon^R = \varepsilon$$
$$(wx)^R = x.(w^R), \quad x \in X,\ w \in X^*.$$

Examples:

$X = \{a, b\}$ is the given alphabet.
$L_1 = \{a, b\}$ is the language over X, containing all words of length 1.
$L_2 = \{a.a, a.b, b.a, b.b\}$ is the language over X, containing only those words, that have length 2.
$L_3 = \{\varepsilon\} \cup L_1 \cup L_2$ is the language over X, containing only words of length less than or equal to 2.
$L_4 = \{x^{2n} \mid x \in X \wedge n > 0\}$ is the language over X, containing only words of even length, which consist of all a's or all b's.

It is obvious, that any subset of the universal language X^* is a language over X, containing only finite words.

2.2 Regular expressions

As was shown in the previous paragraph, languages of finite length words over a given alphabet X are all subsets of the universal language X^*. Therefore, one can define other languages over the same alphabet, using the set-theoretical operations on given languages:

Union:
$L_1 \cup L_2 = \{w \in X^* \mid w \in L_1 \vee w \in L_2\}$
Intersection:
$L_1 \cap L_2 = \{w \in X^* \mid w \in L_1 \wedge w \in L_2\}$
Complement:
$\bar{L} = \{w \in X^* \mid w \notin L\}$

(2.1)

Besides those well-known operations, some others are:

Concatenation:
$L_1 . L_2 = \{w = u.v \mid u \in L_1 \wedge v \in L_2\}$
Power:
$L^0 = \{\varepsilon\}$
$L^n = L^{n-1} . L, \quad n > 0$
or, equivalently,
$L^n = \{w \in L^* \mid |w| = n\}$
Kleene-closure:
$L^* = \bigcup_{0 \leq n < \infty} L^n$
Reverse:
$L^R = \{w^R \mid w \in L\}$

(2.2)

Note, that $\varnothing . L = L . \varnothing = \varnothing$, but $\{\varepsilon\} . L = L . \{\varepsilon\} = L$.

Because X itself is a language over X, the Kleene-closure of X is defined and it is obvious, that $X^* = \bigcup_{0 \leq n < \infty} X^n$ equals the previously described universal language X^*.

Regular languages over a given alphabet X can be constructed by starting with the empty language $\varnothing$ and singleton languages $\{x\}$ for all $x \in X$ and using the operations of union, concatenation and closure.

Example:

$X = \{a, b\}$.
$L = \{w \in X^* \mid |w| = 2 \wedge w[1] = a\}$ can be constructed as follows:

> $A = \{a\}$ and $B = \{b\}$.
> $L_A = A \,.\, A$ and $L_B = A \,.\, B$.
> $L = L_A \cup L_B$.

Regular expressions (RE's) are a way of describing, which operations have to be performed, to construct a certain regular language [5]. The operations of concatenation, closure and union suffice to construct regular languages of finite length words.

The definition of a regular expression over an alphabet X is:

- $\varnothing$ is a regular expression over X.
- if $x \in X$, then x is a regular expression over X.
- if α and β are regular expressions over X, then so are $(\alpha + \beta)$ and $(\alpha \,.\, \beta)$.
- if α is a regular expression over X, then so is α^*.
- only those expressions, formed by using the previous rules are regular expressions.

The language $L(\alpha)$ described by the regular expression α is defined as:

- $L(\varnothing) = \varnothing$.
- if $x \in X$, then $L(x) = \{x\}$.
- if α and β are regular expressions over X, then
  $L(\alpha + \beta) = L(\alpha) \cup L(\beta)$ and
  $L(\alpha \,.\, \beta) = L(\alpha) \,.\, L(\beta)$.
- if α is a regular expression over X, then $L(\alpha^*) = (L(\alpha))^*$.

(2.3)

There may be several expressions for the same language. Two regular expressions for the language L in the last example are:

$a \,.\, a + a \,.\, b$
$a \,.\, (a + b)$

Instead of noting $L = \mathrm{L}(\alpha)$, often a shorthand $L = \alpha$ is used.

Usually, the dot will be omitted, when the symbols are all represented by single characters. The parentheses will sometimes be omitted, according to the priority of the operator signs. The precedence of operators, going from high to low priority, is

$* \quad . \quad +$

which means, that the regular expression

$(a+(((b+a))^*.(b)^*))$

can be written as

$a+(b+a)^*.b^*$.

2.3 Finite automata

A finite automaton (FA) is a quintuple

$M = (Q, X, \delta, I, F)$

where

Q is the finite set of all states M can be in.
X is the finite set of all possible input symbols.
$\delta \subseteq Q \times X \times Q$ describes the transitions.
$I \subseteq Q$ is the set of initial states.
$F \subseteq Q$ is the set of final states.

(2.4)

It is called finite or finite-state, because the number of possible states is finite.

The interpretation is as follows:

The automaton M can be thought of as a mechanism, consisting of a control block capable of being in various states (i.e. the elements of Q) and an input channel, that receives input symbols from the environment (i.e. elements of the input-alphabet X).
The automaton reads input symbols and as a reaction to the symbol read and the current state, M will be in a new state at the next instant of time.
The set of instructions δ describes what will be the next state, depending on the current state and the current input. If δ contains the instruction (q_i, x_r, q_j), this means, that on reading x_r , while M is in state q_i , the automaton will move to state q_j. Thus δ can be interpreted as a function, taking two arguments (the current state and the current input) and returning the corresponding set of next states, i.e.

$$\delta:\ Q \times X \rightarrow P(Q)$$

where

$P(Q)$ is the powerset of Q, i.e. 2^Q.

It follows from this definition, that M may move to more than one state at the same input. This type of automaton is therefore called a non-deterministic automaton. Because M may often be in more than one state at a time, it is more convenient to use another transition function, which takes as its input a set of states, rather than only one state like δ does. Furthermore, the second argument will be a word, rather than a single symbol. This extended transition function is defined as:

$$\begin{aligned}
&\Delta:\ P(Q) \times X^* \rightarrow P(Q)\\
&\Delta(Q_i,\varepsilon) = Q_i\\
&\Delta(Q_i,x) = \bigcup_{q \in Q_i} \delta(q,x) \quad x \in X.\\
&\Delta(Q_i,x.w) = \Delta(\Delta(Q_i,x),w) \quad x \in X,\ w \in X^*.
\end{aligned} \tag{2.5}$$

If δ is defined as a function

$$\delta:\ Q \times X \rightarrow Q$$

and I is a singleton, then M is a deterministic automaton.

The initial and final states of M have the following meaning:

Assume, M is in its initial states and is offered a word w over X at its input channel, one symbol at a time. After reading a symbol x, M moves to the next states, defined by $\Delta(I,x)$. After having read all symbols of w, M is in some subset $Q_f = \Delta(I,w)$ of Q. The word w has taken automaton M from states I to states Q_f. If $Q_f \cap F \neq \emptyset$, the word w is said to be accepted by M, otherwise w is said to be rejected by M.
One can now define a language corresponding to an automaton.
The language accepted by automaton M is the set of all possible words that are accepted by M:

$$L(M) = \{w \in X^* \mid \Delta(I,w) \cap F \neq \emptyset\} \tag{2.6}$$

Because this language is accepted by a finite automaton, such a language is called finite-state. The language accepted by M is also called the behaviour of M.

2.4 Representation of automata by state-graphs

A very useful (graphical) representation of an automaton is in the form of state graphs. An example is given in figure 1. In a state graph, each node represents a state of the

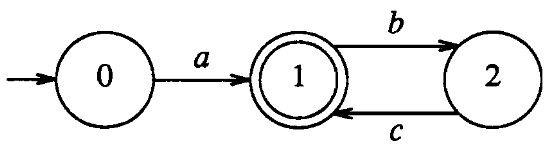


Figure 1. State-graph representation of a FA

automaton. The edges are directed and labelled and these labels represent the input symbols of the automaton. If there is an edge, going from node q_i to node q_j and it is labelled with x_r, then that edge represents one instruction of the transition function δ. To be more precise, it is the instruction (q_i,x_r,q_j). Initial states are marked by an unlabelled arrow pointing to them. Final states are marked by a circle.

A path in a state graph carries a word, i.e. the concatenation of the labels of all edges forming the path. A flow in a graph is any set of paths and obviously a flow carries a language, consisting of all words carried by paths in that flow. Clearly, a word w is accepted by the automaton, if and only if the graph contains a path carrying w, starting at an initial state node and leading to a final state node. Following equation 2.6, the flow from the initial states towards the final states carries the language that is accepted by the represented automaton.

2.5 Silent moves and ε-closures

An automaton can be non-deterministic, if it has more than one initial state or if it has a transition function δ, which allows more than one next state on the same input. There is another reason, why some automata are non-deterministic. Those automata have got the possibility to go to another state, without having read an input symbol. These silent moves (or ε-moves) are represented in δ by instructions of the form:

(q_i,ε,q_j) or $\delta(q_i,\varepsilon) = \{q_j\}$

The symbol ε can be seen as a new symbol, added to the input alphabet of the automaton. As before, it represents the empty word. In a state-graph, silent moves may be represented by edges without a label or labelled with ε.

The ε-closure is defined as:

$\varepsilon\text{-CLOS}(q_i) = \{q_j \mid \text{there is a path from } q_i \text{ to } q_j\text{, labelled } \varepsilon\}$

These paths may have any length, since $\varepsilon^n = \varepsilon$. Of course every state can be regarded as having a path towards itself, labelled ε, because the word carried by such a path is the empty word, which is the same as the word carried by a path of length zero. Therefore ε-CLOS(q_i) always includes q_i.

The definition of ε-CLOS can be extended to sets of states, by using:

$$\varepsilon\text{-CLOS}(Q) = \bigcup_{q_i \in Q} \varepsilon\text{-CLOS}(q_i)$$

The definition of Δ (equation 2.5) is further extended to $\hat{\delta}$:

$$\begin{aligned}
&\hat{\delta}\colon \mathrm{P}(Q) \times X^* \to \mathrm{P}(Q) \\
&\hat{\delta}(Q_i, \varepsilon) = \varepsilon\text{-CLOS}(Q_i) \\
&\hat{\delta}(Q_i, x) = \varepsilon\text{-CLOS}(\Delta(\varepsilon\text{-CLOS}(Q_i), x)), \quad x \in X. \\
&\hat{\delta}(Q_i, x.w) = \hat{\delta}(\hat{\delta}(Q_i, x), w), \quad x \in X,\ w \in X^*.
\end{aligned} \tag{2.7}$$

As before, the language accepted by an automaton with silent moves is:

$$\mathrm{L}(M) = \{ w \in X^* \mid \hat{\delta}(I, w) \cap F \neq \varnothing \} \tag{2.8}$$

It is proved [7] that for every automaton with silent moves, an automaton without silent moves can be found, which accepts the same language.
Without repeating the proof here, it is stated, that for $M = (Q, X, \delta, I, F)$ with silent moves, an equivalent (i.e. accepting the same language) automaton without silent moves is given by:

$$M' = (Q, X, \hat{\delta}, I, F')$$

where

$$F' = \begin{cases} F \cup I & \text{if } \varepsilon\text{-CLOS}(I) \cap F \neq \varnothing \\ F & \text{otherwise} \end{cases}$$

2.6 Relation between deterministic and non-deterministic automata

Because every deterministic finite automaton (DFA) can be thought of as a special case of non-deterministic finite automata (NFA), it is obvious, that for every DFA an equivalent NFA can be given, namely the DFA itself. It is also true, that every NFA is representable by an equivalent DFA. Without giving the formal proof [7], the construction of a DFA from a NFA is given here, because it will be frequently used in FADELA. This construction is known as the subset-construction.

For $M = (Q, X, \delta, I, F)$, an equivalent DFA is given by:

$$M' = (Q', X, \hat{\delta}, i, F')$$

where

$Q' = \mathrm{P}(Q)$
$\hat{\delta}$ is as defined in equation 2.7.
$i = \varepsilon\text{-CLOS}(I)$.
$F' = \{q \in Q' \mid q \cap F \neq \varnothing\}$

(2.9)

It seems, that the number of states in the DFA $|Q'| = 2^{|Q|}$, but in many cases, a lot of states in Q' can never be reached by a flow from i to F', which is the only flow of interest, because it defines the language accepted by M'.

States that are never reached can be deleted, without affecting the language of an automaton. The useful states of $M = (Q, X, \delta, I, F)$ are given by:

$$U = \{q \in Q \mid \exists\, v, w \in X^* : q \in \hat{\delta}(I, v) \wedge \hat{\delta}(\{q\}, w) \cap F \neq \varnothing\} \qquad (2.10)$$

2.7 Automata accepting the language of regular expressions

In the previous paragraphs, a relation between finite automata and languages was given. Equation 2.3 gives the relation between a regular expression and a language. Equation 2.6 and 2.8 state, that every finite automaton represents a language over its input alphabet. It can be proved, that automata and regular expressions represent the same class of languages [7] and therefore, one can view regular expressions as descriptions of the behaviour of automata.

The construction of an NFA with silent moves, that accepts the language of a given regular expression, is as follows and differs from the construction given in [6,7] where only NFA's with single initial and single final states are considered:

The most simple RE's are $\varnothing$, ε (which is equivalent to $\varnothing^*$) and $x \in X$, for which the corresponding NFA's are shown in figures 2,3 and 4 respectively.

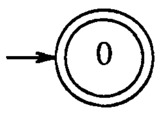


Figure 2. FA for ε

As equation 2.3 shows, regular expressions are formed, using these simple elements and the operations of concatenation, closure and union. These operations can also be applied to NFA's. Assume $M_1 = (Q_1, X_1, \delta_1, I_1, F_1)$ and $M_2 = (Q_2, X_2, \delta_2, I_2, F_2)$ are NFA's accepting the languages of RE_1 and RE_2 respectively. Without loss of generality, Q_1 and Q_2 are considered disjoint, because this can always be achieved by simply renaming

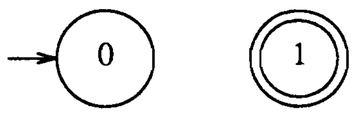


Figure 3. FA for $\emptyset$

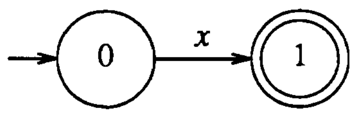


Figure 4. FA for x

the states.

The union $RE = RE_1 + RE_2$ is represented by

$$M = (Q_1 \cup Q_2, X_1 \cup X_2, \delta_1 \cup \delta_2, I_1 \cup I_2, F_1 \cup F_2) \tag{2.11}$$

which obviously accepts both L(M_1) and L(M_2) and is shown in figure 5.

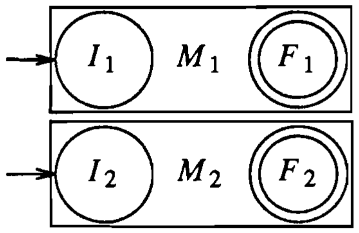


Figure 5. Union of M_1 and M_2

The concatenation $RE = RE_1 \,.\, RE_2$ is represented by

$$M = (Q_1 \cup Q_2, X_1 \cup X_2, \delta, I_1, F_2)$$

where

$$\delta = \delta_1 \cup \delta_2 \cup (f, \varepsilon, i) \text{ for all } f \in F_1 \text{ and } i \in I_2. \tag{2.12}$$

Figure 6 shows this construction and it is easily seen, that any path leading from an initial state of M_1 towards a final state of M_2 carries a word w_1 in M_1, followed by ε and a word w_2 in M_2. Once a silent move from M_1 to M_2 is passed, there is no way back into M_1. The concatenation $w_1.\varepsilon.w_2$ of course yields $w_1.w_2$. The closure $RE = RE_1{}^*$ is in

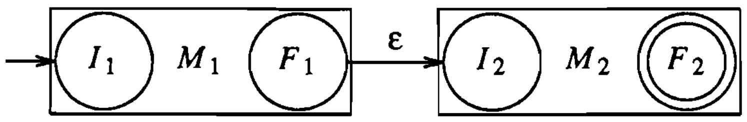


Figure 6. Concatenation of M_1 and M_2

fact the concatenation of zero or more instants of RE_1. The construction in figure 7 represents this repeated concatenation and M is given by

$$M = (Q_1 \cup I, X_1, \delta, I, I)$$

where

$I = \{s\}$, with s being a newly added state, that is both initial and final.
$\delta = \delta_1 \cup (f, \varepsilon, s) \cup (s, \varepsilon, i)$ for all $f \in F_1$ and $i \in I_1$.

(2.13)

By defining the new initial state s, which is also final in the closure-automaton, the possibility of zero concatenations is created, yielding the empty word. Silent moves from the (originally) final states towards this state and from it towards the (originally) initial states, create the repeated concatenation of words of M_1. The state s is the only way to re-enter M_1, once a word of M_1 is passed.

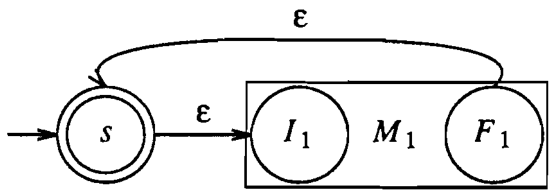


Figure 7. Kleene-closure of M_1

From the given elementary RE's and these operations, all other RE's can be constructed. Clearly, by performing the previously described operations on the corresponding automata, for every RE an equivalent NFA can be constructed.

2.8 Regular expressions describing the languages accepted by automata

Any FA corresponds to a finite state language. Any regular language may be described by a regular expression. It is proved, that the class of finite state languages and regular languages are identical. Again in [6,7] the proof is given for the following construction:

One obvious method, for creating a regular expression of the behaviour of a given FA is starting at every initial state and from there collecting all possible paths towards every final state. This is always correct, but a problem arises, when loops in the state graph are encountered. For example, in the simple FA given in figure 1, this method would lead to an expression like:

$$RE = a+a.b.c+a.b.c.b.c+a.b.c.b.c.b.c+.....$$

The Kleene-closure which is obvious the cause of any loop, should be used to describe this repetition:

$$RE = a.(b.c)^*$$

A better way to construct the expression for the flow in $M = (Q, X, \delta, I, F)$ is:

Let n be the number of states in $Q = \{q_1, .., q_n\}$.
For $1 \le i \le n$ and $1 \le j \le n$ define the regular expression α_{ij}^0 by:

$$\alpha_{ij}^0 = \sum_{x \in X} x \text{, such that } q_j \in \delta(q_i, x)$$
$$\alpha_{ij}^0 = \varnothing \text{ if there is no } x \in X \text{, such that } j \in \delta(i, x)$$
$$q_i \in \delta(q_i, \varepsilon) \text{, therefore always } \varepsilon \in \alpha_{ii}^0$$

(2.14)

This means, that α_{ij}^0 is the regular expression for the paths, going from i to j, without crossing any state > 0, where crossing means both entering and leaving. No states > 0 simply means, that α_{ij}^0 only describes the single symbols between i and j. Recursively, the next 'generations' of expressions (for $0 < k \le n$) are defined as:

$$\alpha_{ij}^k = \alpha_{ij}^{k-1} + \alpha_{ik}^{k-1} . (\alpha_{kk}^{k-1})^* . \alpha_{kj}^{k-1}$$

(2.15)

Now α_{ij}^k describes the paths from i to j passing through states $\le k$ and thus α_{ij}^n describes all possible paths from i to j. The regular expression describing the language accepted by M is now given by:

$$RE = \sum_{\substack{i \in I \\ f \in F}} \alpha_{if}^n$$

(2.16)

In computing the successive generations of expressions, some simplifications can be made. For $k = i$

$$\alpha_{ij}^k = (\alpha_{ii}^{i-1})^* . \alpha_{ij}^{i-1}$$

and for $k = j$

$$\alpha_{ij}^k = \alpha_{ij}^{j-1} . (\alpha_{jj}^{j-1})^*.$$

For the automaton in figure 1, a regular expression corresponding to the accepted language is:

$$\alpha_{01}^{2} = \alpha_{01}^{1} + \alpha_{02}^{1} \, . \, (\alpha_{22}^{1})^{*} \, . \, \alpha_{21}^{1} = a + a.b \, . \, (c.b)^{*} \, . \, c$$

where α_{ij}^{1} was obtained by inspection of the state-graph, but could also be computed by further recursion following equation 2.15.

3. *FINITE AUTOMATA AND ω-REGULAR EXPRESSIONS*

One conclusion of the previous chapter is, that every regular language is finite state. This is true for languages containing only finite length words. However, not only finite length words can be accepted by finite automata, but also infinite words. Of course the accepting conditions used before are incapable for infinite words, because a path from initial to final states is always finite.

To make an infinite path through an automaton (having only a finite number of states), some states have to be passed infinitely many times. The only way for a path to re-enter states is to contain a loop. In the automata discussed before, loops corresponded to the Kleene-closure, which is by definition only a finite repetition.

The structure of finite automata seems useful, to accept infinite words, but the accepting conditions will have to be adjusted and the meaning of loops has to be specified more carefully.

In this chapter, the theory of regular expressions is extended to ω-regular expressions, such that a class of languages consisting of infinite-length words can be described by them. Then two types of automata, capable of accepting languages defined by ω-regular expressions are dealt with.

3.1 ω-words and ω-languages

X is again an alphabet over which a language is defined.
The words, called ω-words, are constructed as before, but all have infinite length, denoted by:

$$|w| = \omega$$

An ω-language is a set of ω-words.
The set of all possible ω-words over X is denoted by X^{ω}.

Concatenation of words and ω-words is defined and always has an ω-word as a result:

For $u \in X^*$ and $v \in X^{\omega}$, $w = u.v$ is a ω-word, for which

$$|w| = \omega$$
$$w[i] = \begin{cases} u[i] & 0 < i \leq |u| \\ v[i-|u|] & |u| < i \end{cases}$$

Obviously, concatenating anything after $w \in X^{\omega}$ is of no use. Therefore, $u.v$ for $u \in X^{\omega}$ and $v \in X^{\omega} \cup X^*$ by definition equals u.

3.2 ω-regular expressions

The closure operation is extended by defining the strong iteration closure or ω-closure as:

$$w^{\omega} = w.w.w.... \tag{3.1}$$

which means the infinite repetition of w.

Every ω-word looks like

$w = \alpha\,.\,\beta^{\omega}$

where

$\alpha \in X^*$
$\beta \in X^* - \{\varepsilon\}$

ω-regular expressions have the same form, where α and β are both normal regular expressions, but $\beta \neq \emptyset$.

Not every infinite word over X can be described by ω- regular expressions, as the next example shows:

$w = 0.1.0.0.1.0.0.0.1.0.0.0.0.1...$

However, it is proved, that ω-regular languages can be represented by finite automata and every finite-state language can be described by an ω-regular expression [9].

3.3 Finite automata accepting ω-regular languages

Given an ω-word w, a run ρ of automaton M for w is defined as the sequence of states, that M goes through on reading w. Of course, when M is non-deterministic, there may be several paths through M for w and thus there are more runs possible for w. Because every run must have infinite length and the number of states is finite, some states have to be passed more than once. In fact, to have an infinite run, there have to be states, that occur infinitely many times in that run.
The set of states, that occur infinitely many times in a run ρ is denoted by INF(ρ).

In automata accepting only finite words, the accepting condition was based on the states that M was left in after reading w. Automata accepting infinite words do not halt in a specific state, but they keep moving around in INF(ρ) while reading the part β^{ω} of $w = \alpha\,.\,\beta^{\omega}$.
By defining various accepting conditions based on the sets INF(ρ), different classes of languages can be accepted by finite automata [10,11]. In this report, only two classical types are discussed, namely B- and M-automata.

3.3.1 B-automata

One of the frequently used accepting conditions is the one proposed by Büchi [12]. Although currently not implemented in FADELA, Büchi automata (denoted B-automata) are given some attention here, because FADELA can easily be extended to hold this kind of automata too.

The structure of B-automata is the same as for the automata discussed in chapter 1. Thus

$M = (Q, X, \delta, I, F)$

Only the accepting condition is changed:

A run ρ corresponds to an accepted ω-word, if and only if

$$\mathrm{INF}(\rho) \cap F \neq \varnothing \tag{3.2}$$

Thus an accepted ω-word is recognized on a B-automaton, if the loop at the end of the run for that word contains at least one state in F.

The definition of the language accepted by a B-automaton B is:

$$\mathrm{L}(B) = \{w \in X^{\omega} \mid \exists \rho \text{ for } w\text{: } \mathrm{INF}(\rho) \cap F \neq \varnothing\} \tag{3.3}$$

The known equivalence between deterministic and non-deterministic automata accepting finite words (equation 2.9), is not valid for B-automata, as can be seen from the example of B in figure 8 and the result of the subset-construction on it, given as B_{det} in figure 9.

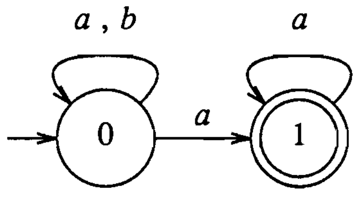


Figure 8. Non-deterministic B-automaton B

It is easily observed, that $\mathrm{L}(B) = (a+b)^* \,.\, a^{\omega}$. In the deterministic version however, a proper choice for the final (or accepting) states still has to be made. But none of the possible accepting sets { }, {{0}}, {{0,1}} or {{0},{0,1}} causes B_{det} to accept the same set of ω-words, all ending in infinite repetition of a, as does B.

This points out the great disadvantage of B-automata:
Not for every ω-regular language an accepting deterministic B-automaton can be constructed, using the subset construction. It is proved, that there are non-deterministic B-automata, for which there exist no deterministic equivalents [4].

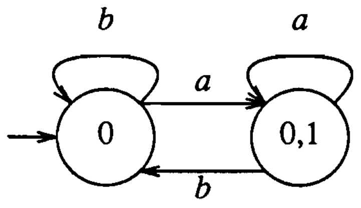


Figure 9. Deterministic B-automaton B_{det}

3.3.2 M-automata

Another kind of automata, proposed by Muller [12] is implemented in FADELA, because for M-automata there is an equivalence between deterministic and non-deterministic versions. This makes M-automata more attractive than B-automata:
For every ω-regular language, an accepting deterministic finite automaton exists [11]. Having deterministic automata makes taking the complement easy.

A M-automaton is defined as

$M = (Q, X, \delta, I, C)$

where

$C \subseteq \mathrm{P}(Q)$ is a set of limitsets.

(3.4)

The accepting condition for M-automata is based on these limitsets. A run ρ corresponds to an accepted ω-word, if and only if

$\mathrm{INF}(\rho) \in C$

(3.5)

Thus an accepted ω-word is recognized on a M-automaton, if the loop at the end of the run is exactly one of the limitsets given in C.

The language accepted by a M-automaton M is:

$\mathrm{L}(M) = \{w \in X^{\omega} \mid \exists\rho \text{ for } w\text{: } \mathrm{INF}(\rho) \in C\}$

(3.6)

C is always a subset of P(Q) but not all elements of P(Q) can be limitsets.
Because a limitset is a set of states in which the run ρ of an accepted ω-word passes every state infinitely many times, there must be at least one loop in it, containing all these states.
In other words, a limitset must always be the set of states in a loop in the state-graph that represents the automaton.

As a result, some states may be deleted, without affecting the language accepted by the M-automaton. The only states that are useful are, assumed that C only contains legal limitsets:

$$U = \{q \in Q \mid \exists u,v \in X^*, c_i \in C : q \in \hat{\delta}(I,u) \wedge \hat{\delta}(\{q\},v) \cap c_i \neq \varnothing\} \tag{3.7}$$

3.4 M-automata accepting the language of ω-regular expressions

For every ω-regular expression, a non-deterministic Muller-automaton accepting the language described by that expression, can be constructed. This can be done in exactly the same way as was done in the previous chapter for automata accepting finite words, by performing the operations of union, concatenation and closure on automata, but here the new operation of the ω-closure has to be performed also.

The union-operation (equation 2.11) must be adjusted, to work on pairs of M-automata. A union between a M-automaton and a 'normal' one may never occur, since that would mean, that the resulting automaton accepts both finite and infinite words.

The union $RE = RE_1 + RE_2$, where RE_1 and RE_2 both are ω-regular expressions and are represented by M-automata M_1 and M_2 respectively, is represented by:

$$M = (Q_1 \cup Q_2, X_1 \cup X_2, \delta_1 \cup \delta_2, I_1 \cup I_2, C_1 \cup C_2). \tag{3.8}$$

An extension for the concatenation-operation (equation 2.12) also has to be made. A concatenation will either combine two 'normal' automata, in which case equation 2.12 is used, or will put a M-automaton behind a 'normal' one:

$$M = (Q_1 \cup Q_2, X_1 \cup X_2, \delta, I_1, C_2) \tag{3.9}$$

where

$$\delta = \delta_1 \cup \delta_2 \cup (f, \varepsilon, i) \text{ for all } f \in F_1 \text{ and } i \in I_2.$$

The new operation of ω-closure always takes an automaton M_1 accepting finite words. The resulting automaton is a M-automaton M, which must accept $(\mathrm{L}(M_1))^{\omega}$. To do this, a proper construction of limitsets has to be made [3].

Constructing the ω-closure is the same as for the Kleene-closure (equation 2.13). The limitsets are all those sets, that contain the initial/final state s:

$$M = (Q_1 \cup I, X_1, \delta, I, C)$$

where

$I = \{s\}$, with s being the newly added state.
$\delta = \delta_1 \cup (f, \varepsilon, s) \cup (s, \varepsilon, i)$ for all $f \in F_1$ and $i \in I_1$.
$C = \{c \in \mathrm{P}(Q) \mid s \in c\}$.

(3.10)

As mentioned before, not all elements of C are useful limitsets. They have to be loops and clearly, all loops that contain s are precisely all the loops going from s through M_1 and back to s. (Recall figure 7.) Thus these loops describe all possible repetitions of words of M_1.

In state-graph representation of M-automata, there is no general way to clearly show, which subsets of Q are limitsets. The 'best' way to overcome this problem, is to mention C along with the picture.

3.5 Language represented by M-automata

The class of ω-regular languages equals the class of languages accepted by M-automata [11]. In the previous paragraph, one part of the proof is given, by describing the method for creating a non-deterministic M-automaton, accepting the language of an ω-regular language. The converse can also be proved by a constructive method [3,9,11].

Equation 3.6 states, that the language accepted by a M-automaton M consists of those ω-words, that lead M from its initial states towards one of its limitsets and then forever repeats some path inside that limitset, passing all states in it. Thus every word accepted looks like:

$$w = \alpha.\beta^{\omega}$$

where

α is the word that takes M from some initial state towards a state b inside a limitset c.
β is a word that is carried by a loop inside c, starting at b, passing all states in c.

The language accepted by M can now also be described as the union of all this ω-words w:

$$RE = \sum_{c_i \in C} E_i$$

where

$E_i = \alpha\,.\,(\beta_{1,2}.\beta_{2,3} \cdots \beta_{k,1})^{\omega}$.
$c_i = \{s_1, s_2, .., s_k\}$.
α is the set of words, that take M from its initial state towards s_1.
β_{ij} is the set of words, that take M from s_i towards s_j, without passing any state outside of c_i.
These α and β can be computed by using equations 2.14 and 2.15.

(3.11)

Using this method, RE will give a description of the ω-language accepted by M-automaton M.

3.6 Relation between deterministic and non-deterministic M-automata

It is proved [3,11], that the class of ω-languages accepted by deterministic M-automata equals the class of ω-languages that non-deterministic M-automata accept. Unfortunately, no construction is given for creating a deterministic M-automaton equivalent to a non-deterministic one. Only constructive proofs are given, which use the construction of ω-regular expressions as an intermediate step.

The subset-construction (equation 2.9) however can be used, but the right limitsets have to be constructed in the deterministic version. In this paragraph, my method for directly constructing deterministic M-automata from non-deterministic ones is given.

As an example, consider the NFA in figure 10 and its equivalent deterministic graph in figure 11. When the only limitset is $\{1\}$, the ω-language accepted by M is $a.(b^*.b.a)^*.b^{\omega}$, as can easily be observed, using equation 3.11.

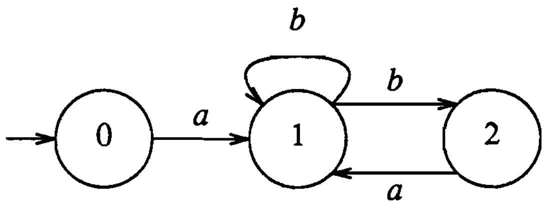


Figure 10. M-NFA M with limitset $\{1\}$

The limitset $\{1\}$ is projected onto $\{<1>,<1,2>\}$ by the subset-construction, but taking this set to be the limitset for M_{det} would also cause the ω-words described by $a.(b.b^*.a)^{\omega}$ to be accepted by M_{det}. The reason for this error is the following:

The subset-construction maps every path (or loop) of the NFA onto a path (or loop) in the DFA.
The path 1-b-1-b-1-... is mapped onto <1>-b-<1,2>-b-<1,2>-...
The path 1-b-2-a-1-... is mapped onto <1>-b-<1,2>-a-<1>-...
Both paths are mapped onto the same set of states $\{<1>,<1,2>\}$. The first path

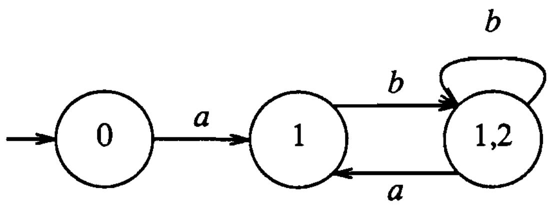


Figure 11. M-DFA M_{det}

represented an ω-word, because it never left the limitset {1}, but the second did not stay within the limitset. After mapping by the subset-construction however, both paths are no longer distinguishable by their sets of passed states.

The solution to this problem is making sure, that every limitset is disjoint with respect to paths that are entering the limitset, but later leave it.

Making limitsets disjoint from other loops can be done by duplicating the limitsets and defining the duplicates to be the limitsets instead of the original ones. Duplicating $c = \{q_1,..,q_k\}$ means:

$$\begin{aligned} &Q = Q \cup \{q'_i \mid q_i \in c\} \\ &\delta = \delta \cup (q_i,x_r,q'_j) \cup (q'_i,x_r,q'_j), \quad (q_i,x_r,q_j) \in \delta,\ q_i,q_j \in c. \end{aligned} \tag{3.12}$$

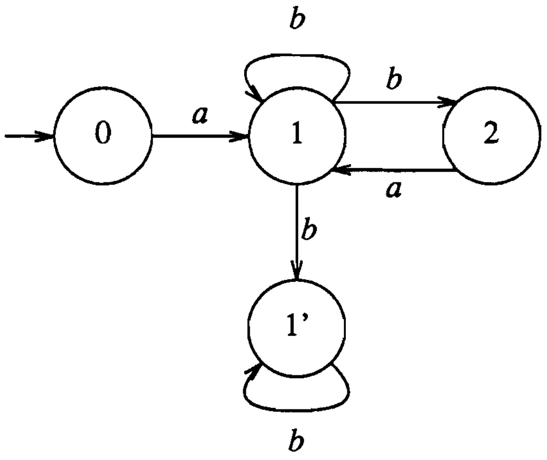


Figure 12. M-NFA M' after duplication of {1}

In this way, all paths in c now have a twin in $c' = \{q'_1,..,q'_k\}$ and at any state in c, the next state on a certain input may be either in c or in c'. The result of this duplication for the NFA in figure 10 is given in figure 12. Clearly, the duplication method isolates the limitsets, because there is no edge leaving a limitset. Thus there can never be a path entering a limitset, but not staying in it.

It is not difficult, to show that duplicating the limitsets does not affect the language accepted by M.
Without loss of generality, assume there is only one limitset, thus $C = \{c\}$, since otherwise C is just a union and the class of ω-regular languages is closed under union.
Let M' be the resulting automaton after duplicating the limitset c of M. Let ρ be a run for an ω-word accepted by M. Since there are no edges removed, every path in M also exists in M'. At some instant, ρ on M enters c (at state q_0) and from then on will stay in c. (i.e. α-part ended and β^ω started) The corresponding ρ' on M' can of course also enter c (at the same state q_0) on M'. The next move on M will be from one state in c to a next state q_1 in c. M' can make the corresponding move to state q'_1 in c', because for every move from a state $q_0 \in c$ towards $q_1 \in c$ duplication created a move from $q_0 \in c$ towards $q_1 \in c'$. All subsequent moves on M within c have corresponding moves on M' within c'.
Thus $L(M) \subset L(M')$.

The converse goes quite the same. Let ρ' be a run for an ω-word accepted by M'. At some time, ρ' moves on M' from a state $q_0 \notin c'$ to a state $q_1 \in c'$, thus starting the β^ω-part of the word, because there are no edges leaving c'. Because every move towards a state $q' \in c'$ on M' corresponds to a move towards $q \in c$ on M, it is clear, that M can enter its limitset too. The only possible subsequent moves on M' are all within c' and because of the duplication, all these moves of course have corresponding moves within c on M.
Thus $L(M') \subset L(M)$, which completes the proof that $L(M') = L(M)$.

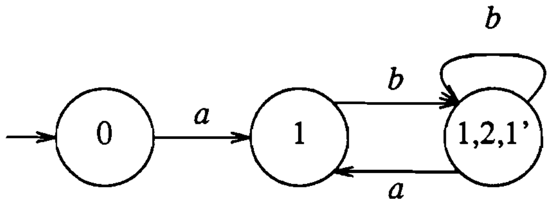


Figure 13. M-DFA M'_{det} with limitset {<1,2,1'>}

As shown in figure 13, determinization of the NFA after duplication yields a DFA, in which the limitset {<1,2,1'>} corresponds to these sets onto which limitset {1'} is mapped. Thus for every limitset c of M_{NFA}, the corresponding limitset c' of M_{DFA} is:

$$c' = \{q \in P(Q) \mid q \cap c \neq \emptyset\} \tag{3.13}$$

Of course, no edge may occur between states of such a limitset, if there is no corresponding edge (i.e. carrying the same label) within the original limitset. It is now proved, that such an illegal edge can never occur, if the limitsets of the NFA are isolated (using duplication) before performing the subset construction and constructing DFA-limitsets using equation 3.13.

Let $M = (Q, X, \delta, I, C)$ be the NFA, where C contains only one limitset c which is the result of duplication.
Let $M' = (Q', X', \delta', I', C')$ the DFA, constructed via the subset-construction. Clearly C' is also singleton.
Now take any $(Q_1, x, Q_2) \in \delta'$, with $Q_1 \in C'$ and $Q_2 \in C'$. Q_1 consists of states in Q, which are all reachable in M from some state, by the same word.
If $Q_1 \cap c = \varnothing$, then $Q_1 \notin C'$ and this implies a contradiction. Thus it follows $Q_i \cap c \neq \varnothing$.
If $\exists q_0 \in Q_1 \cap c : \delta(q_0, x) \neq \varnothing$ then of course $Q_2 \in C'$ and the edge is legal, because it has its corresponding edge within the limitset of M.
Else $\forall q_0 \in Q_1 \cap c \;\; \delta(q_0, x) = \varnothing$. In that case, if $\forall q \in Q_1 \;\; \delta(q, x) \cap c = \varnothing$, then $Q_2 \notin C'$, again a contradiction.
Now take $q_1 \in Q_1$ and $q_1 \notin c$ and $(q_1, x, q_2) \in \delta$, with $q_2 \in c$. This implies, that this edge will occur in the limitset c' of the DFA M', while it was not in the limitset c of the NFA M. However, the duplication ensures, that the only edges pointing towards a state $q_2 \in c$ must come from a state $q_1 \in c$ (contradiction) or else, an equivalent edge will exist within the limitset, i.e. $\exists q_3 \in c : (q_3, x, q_2) \in \delta$ and this q_3 exists because it is the duplication of q_1. It also follows from the duplication, that q_1 and q_3 are reachable from the initial states of M, by the same word. That however would finally imply, that there is a corresponding edge for (Q_1, x, Q_2) within the original limitset c.
That completes the proof, that there can be no 'new' edge introduced in the limitset of the DFA, when the duplication method is used, before the subset-construction.

As mentioned before, not all sets of states are legal limitsets. Therefore, only those limitsets, that are constructed using equation 3.13 and that each form a loop, form the set of limitsets of the DFA.

The deterministic version of M-automaton $M = (Q, X, \delta, I, C)$ can now be given by:

$$M' = (Q', X, \hat{\delta}, i, C')$$

where

$$\begin{aligned} &Q' = \mathrm{P}(Q) \\ &\hat{\delta} \text{ is as defined in equation 2.7.} \\ &i = \varepsilon\text{-CLOS}(I) \\ &C' = \{c' \in Q' \mid c' \text{ as defined in equation 3.13 forms a loop } \} \end{aligned} \tag{3.14}$$

4. FADELA

In this chapter, the Finite Automata DEbugging LAnguage will be described. For the exact syntax and semantics, refer to appendix A. First the need for a tool like FADELA will be explained and the specifications listed. Then some theory and algorithms behind the most important standard functions implemented in FADELA will be given.

4.1 Specifications for a finite automata debugging tool

A useful formal method for verification of protocols or systems is the use of finite-state models. The theory of finite automata offers several proven ways for checking equivalence between various models or checking whether a model accepts certain behaviours. For example:

M is an automaton, the model to be checked. This may, for instance be a mechanism, detecting certain sequences of input-signals. L is a description of the sequences, that M must be able to detect. This may be a regular expression. A way to prove that M indeed detects only sequences in L, is to construct an automaton $M_L = \mathrm{M}(L)$, whose behaviour is described by L. As shown in chapter 1, this can always be done. The complement C_L of this automaton M_L will then accept all sequences, except those described in L. Theory states, that the product-automaton of two automata accepts only those sequences, that are accepted by both components of the product. Therefore, the product $P = M \times C_L$ will accept all sequences, that are accepted by M but not by M_L. Now, if P accepts any sequence, it is clear, that M was not correct, because such a sequence can not be in L. Of course, to prove that M accepts precisely every sequence in L, the product of M_L with the complement of M should also be empty.

The way of testing correctness, as used in this example, is only one of several and there may be many different definitions of correctness. One program, capable of checking behaviours, like was done in the example, will therefore often not be flexible enough. A more useful approach would be, to develop a tool, by which standard functions like product and expression-automaton conversion can be performed. The user can then perform more complex tests, by combining these standard operations.

A useful tool should match the following specifications:

- Possibility to describe finite automata.
- Possibility to describe regular expressions.
- Conversion between these two descriptions of finite state models.
- Simple user interface. This includes the possibility to describe the models in an easy way. The presentation of the models and results of the operations must be easy to understand by the user. This means for instance simplification of regular expressions.

- The most important operations on automata must be standard functions, which can be easily called by the user. These functions include:
    - Union.
    - Concatenation.
    - Closure.
    - Complement.
    - Reflection.
    - Minimization.
    - Determinization.
    - Products, like parallelization, lock-step product and intersection.
- Automata accepting infinite words and ω-regular expressions describing them should also be possible.
- Working with the tool can be done interactively or by using batch-files.
- To make powerful batch-files, control statements and types like integers and booleans should be available.
- Symbols to be used in regular expressions and in transition functions of automata, should be free to choose, not restricted to single characters and digits. Because many transitions may be described by boolean functions, the symbols should really be boolean functions.

The result of trying to implement these specifications is FADELA, a Finite Automata DEbugging LAnguage, which is a complete language, including as standard types: finite automata, regular expressions, integers, booleans, sets of states and words consisting of boolean products. Several functions on these types are implemented, to create a flexible environment for examining finite-state models.

4.2 FADELA interpreter

To create a flexible user-interface, a syntax is defined and an interpreter is build, using the tools LEX and YACC. This syntax (appendix A) describes the form every command given by the user should have. These commands can be:

- Assignments to give a variable a specific value, which may be a constant value, the value of a variable or the value of an expression. An expression is build from the standard functions in FADELA.
- Print statements to show the values of variables, the names of used variables or to show strings or results of computations (i.e. the value of expressions).

- Redirection statements to force output to be send to a specific file or to read next commands from a file (i.e. start batch processing).
- Repetitive (WHILE) and conditional (IF .. ELSE or ON) statements, to control the execution of compound statements.
- Commands to dispose variables or quit the interpreter.

Execution of these statements, including computation of expressions, could have been done 'on the fly', thus having the results available at the time a statement or expression was read. Although this is a very easy and quick way of interpreting a task, problems would arise, when control statements were used, because then the WHILE- or IF-part should have to be re-read. To overcome this, a parsetree is build for every statement and its substatements. If a statement on the highest level is read completely, it is executed, which may involve execution of substatements. Of course, (nested) WHILE- or IF-statements are no longer a problem, because they occur in the parsetree as a pair:

Control statement =
<parsetree for boolean expression , parsetree for substatement>

There are no variable-declarations in FADELA. A variable is created when an assignment to it is made and this assignment automatically defines the type of the variable.
Because expressions of different types may look the same, the result type of the expression is defined by the type of the variables occurring in it. Therefore, the silent variable-declaration takes place, while generating the parsetree and before the execution or computation of the result of the expression.
Variables are stored and retrieved using one general set of functions. All names have to be unique and once a name is used for a variable of a certain type, the same name can not represent a variable of another type, unless the previous variable was disposed. The possible types, all having their own specific data-structure, are:

- Finite automata, either M-automata or automata accepting only finite words.
- (ω-)Regular expressions.
- Integers.
- Booleans.
- Sets of states.
- Finite words, being sequences of boolean products.

Examples:

*bool_* 1, *a* and <*a*,~*b*,*x*> are boolean products,
although *bool_* 1 and *a* each contain only 1 atomic boolean.

{*state* 0, 1 , *X*_ 123} is a set of states.

[*a* *<a,˜b,x>* *a* *a*] is a finite word of length 3.

(*a*+(*bool*_ 1.*<a,b>*)*).a^ω is an ω-regular expression.

4.2.1 Data structure for finite automata

Considering the operations to be performed most frequently on finite automata, a useful data structure is chosen. This structure should make operations like concatenation, union, closure and ω-closure easy, because these operations will be performed while generating a FA for a given regular expression. Furthermore, given current states and an inputvector (i.e. a boolean product), the next states must be computed. While constructing a regular expression for the language accepted by a FA, all edges between given states must be obtained and of course the labels (boolean products) on the edges.

A finite automaton in FADELA is stored in a structure containing:

I = set of initial states.
F = set of final states (in the case of a 'normal' automaton).
or
C = set of limitsets (in the case of a M-automaton).
L = set of all labels on the edges (i.e. possible inputs).
M = matrix describing δ.

where

$M[i][j]$ = sum of the labels of all edges from state i to state j.

An input is always a vector or product of boolean variables, e.g. $x.\bar{z}$ meaning x is true and z is false. Every edge in the automaton may have such vectors as a label, which then describe a boolean function which has to be true, in order to enable the transition along that edge. Thus $M[i][j]$ contains the union of some functions, e.g. $x+y.\bar{z}$ meaning that there is an edge labeled x, going from i to j and there is one labeled $y.\bar{z}$. Clearly, this automaton will move from i to j on input x, because $M[i][j]$ becomes true when x is true.
Now $\delta(i,x)$ is computed by simply taking row $M[i]$ and computing the boolean function at every column for the given value x:

$$\delta(i,x) = \{ j \mid M[i][j](x) = \text{true} \}$$

The operation of union (equation 2.11 and 3.8) is simply summing the matrices. This summing in fact means putting the two matrices together, since the sets of states are disjoint. Concatenation (equation 2.12 and 3.9) involves summing the matrices and filling $M[f][i]$ with the function ε, which of course represents the silent moves. Closures (equations 2.13 and 3.10) are created by adding a new row and column s to the matrix and filling $M[s][i]$ and $M[f][s]$ with ε. Besides these simple combinations of the matrices, the sets I, F, C and L have to be adjusted in the way described by the equations mentioned. These are also simple unions, except for the computation of C in equation 3.10. This involves generating all possible subsets of Q and only keeping those, that are loops containing s.

4.2.2 Data structure for regular expressions

The choice for a useful data structure for regular expressions is very straight forward. These expressions can best be represented by a tree structure, where the nodes look like:

Lab = label (in the case of a leaf node).
Op = operation (in the case of an interior node).
Sub = the first child node.
Next = the next child node of the parent node.

where *Op* can be one of

Union, having more than 1 child node.
Concatenation, having more than 1 child node.
Closure, having only 1 child node.
ω-Closure, also having 1 child node.

In figure 14, a tree for the expression $a.(b+c)^*.d$ is shown.
Operations on RE's, for instance constructing a finite automaton from a given RE, involve recursive descending of the tree structure and performing the operation *Op* on the FA's resulting from the subtrees.

4.3 Operations on finite automata

Besides the operations on automata discussed in chapters 2 and 3, namely concatenation, union, closure and ω-closure, which are standard functions used in generating FA's from RE's, some other operations are standard functions in FADELA.

4.3.1 Complementation

In the introductions to this report and to this chapter, the use of complementation is already mentioned.
Let M be an (M-)automaton accepting language L over alphabet X. Then the complementary (M-)automaton M_c accepts all (ω-)words over X, that are not accepted by M.

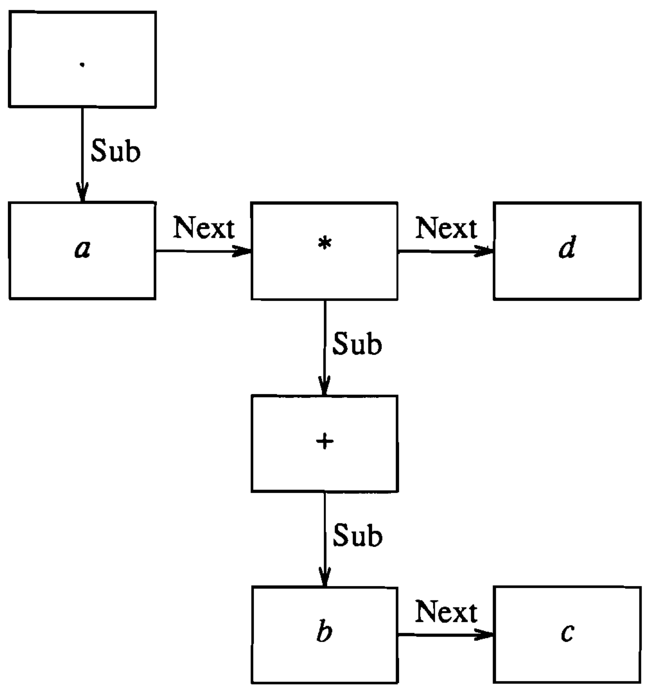


Figure 14. tree-representation of $RE = a.(b+c)^*.d$

In the case of automata accepting only finite words, the complementation is very straight forward:

Assume M is deterministic and δ is fully specified, i.e. for all states q and all symbols x, exactly one next state is defined by $\delta(q,x)$. As mentioned in paragraph 2.6, it is always possible to construct a deterministic automaton equivalent to a given non-deterministic one. Making δ fully specified is done by adding an extra state q_{error} to M and then adding transitions (q,x,q_{error}) for all q and x, where $\delta(q,x) = \emptyset$. This does not influence the language accepted by M, since q_{error} is no final state and has no edges to other states than q_{error} itself. Under these assumptions, there is a path in M for every possible word over X. The complement M_c of M is now constructed by complementing the set of final states F with respect to the set of all states Q. Clearly no word w accepted by M can be accepted by M_c, because the state that w takes M to is not a final state in M_c. Conversely, any word that takes M to a non-final state, takes M_c to a final state.

This simple complementation only works properly for deterministic automata. In a non-deterministic automaton, two paths carrying the same word w may exist, one leading to a final state and the other to a non-final state. Because of the first path, w is accepted by M, but the second path causes M_c to accept the same word w.

For M-automata, the same construction can be used, again under the assumption that M is deterministic and fully specified. Now the set of limitsets C is complemented with respect to the set of all possible loops in M.

4.3.2 Minimization

For the same language, several accepting automata may exist. A minimal DFA is one that has the minimal number of states. Constructing the minimal equivalent of a given DFA is done by combining equivalent states into one state. Two states s and t are equivalent, if the language carried by the flow from s is exactly the same as the language carried by the flow from t. In other words, s and t are equivalent if both are either final or non-final and for every symbol x the next states $\delta(s,x)$ and $\delta(t,x)$ are equivalent.
This leads to the following method to construct a minimal DFA from a given DFA:

Assume δ is fully specified.
First a partition is made, splitting the set of states Q into two not equivalent subsets, namely the final and the non-final states.
Consider one element s of the partition. If the next states $\Delta(s,x)$ are in only one element t of the partition for each symbol x, then the states in s are considered equivalent, else s must be split into subsets that do have next states in only one element t for each symbol.
This splitting leads to successive refined partitions, until no more elements s of a partition need to be split. Thus this final partition is $\pi = \{s_1,..,s_n\}$, such that $\Delta(s_i,x) \subseteq s_j$ for all $x \in X$ and $s_i,s_j \in \pi$.

The minimal DFA is now given by

$$M' = (Q',X',\delta',I',F')$$

where

$Q' = \{1,..,n\}$ representing the elements of π.
$\delta' = \{(i,x,j) \mid \exists(s,x,t) \in \delta : s \in s_i \wedge t \in s_j\}$
$I' = \{i \mid s_i \cap I \neq \varnothing\}$
$F' = \{i \mid s_i \cap F \neq \varnothing\}$

For a DFA with n states and m input symbols, this method has time bound $m.n^2$, forming at most n successive partitions, testing n states on m inputs in constructing each partition. An alternative algorithm is developed by Hopcroft, taking only $m.n \log n$ time [14,15]. The latter algorithm is implemented in FADELA.

For M-automata, no minimization is available in FADELA, because no global minimization algorithm has been found, due to lack of time. However, some decrease in the number of states may be achieved, by using the mentioned algorithms, taking as initial partition all (disjoint) limitsets and the set containing all other states. Thus

$$\pi = C \cup \{q \mid \forall c \in C : q \notin c\}$$

4.3.3 Product operations

Three different product operations are implemented in FADELA, denoted by Parallelization, Lock-step Product and Product, differing in the amount of synchronization between the parts of the product-automaton.

Any product M of automata $M_1 = (Q_1,X_1,\delta_1,I_1,F_1)$ and $M_2 = (Q_2,X_2,\delta_2,I_2,F_2)$ is given by:

$M = (Q,X,\delta,I,F)$

where

$Q = \{(q_1,q_2) \mid q_1 \in Q_1 \wedge q_2 \in Q_2\}$
$I = \{(i_1,i_2) \mid i_1 \in I_1 \wedge i_2 \in I_2\}$
$F = \{(f_1,f_2) \mid f_1 \in F_1 \wedge f_2 \in F_2\}$
X and δ depent on the type of product.

Parallelization
In the case of parallelization, there is no synchronization between the parts of the product. The two automata are simply put together and work independently, although the input channels may coincide when both automata have input symbols in common.

$X = \{<x_1,x_2> \mid x_1,x_2 \in X_1 \cup X_2\}$

Thus every input may contain one input for M_1 and one for M_2, but x_1 may also be equal to x_2 and some inputs may be accepted by only one of M_1 or M_2.

$\delta = \{((s_1,s_2),x, (t_1,t_2))\}$

where $x = <x_1,x_2>$ and either

$t_1 \in \delta_1(s_1,x)$ and $t_2 \in \delta_2(s_2,x)$, thus both parts have a transition
or
$t_1 \in \delta_1(s_1,x)$ and $t_2 = s_2$, thus only M_1 moves
or
$t_1 = s_1$ and $t_2 \in \delta_2(s_2,x)$, thus only M_2 moves.

This product automaton accepts words, that are the result of merging words accepted by M_1 and M_2 respectively. Example:

a.b.b.c is accepted by M_1 and *c.b.d.a* is accepted by M_2. Then for instance *<a.c >.b.<b.d >.<a.c >* and *<a.c >.b.d.b.a.c* are accepted by the Parallelization of M_1 and M_2. Note that *<a.c >* represents a vector that is one input, just like *b*.

Lock-step Product
In the case of Lock-step Product, the synchronization between the parts ensures that both parts have a transition on every input, although not necessarily due to the same part of the input.

$$X = \{<x_1,x_2> \mid x_1,x_2 \in X_1 \cup X_2\}$$

This is the same as for Parallelization.

$$\delta = \{((s_1,s_2),x, (t_1,t_2)) \mid t_1 \in \delta_1(s_1,x) \wedge t_2 \in \delta_2(s_2,x) \}.$$

This product automaton accepts words, constructed by placing in parallel words of equal length accepted by M_1 and M_2. Example:

a.b.b.c is accepted by M_1 and *c.b.d.a* is accepted by M_2. Then $<a.c>.b.<b.d>.<a.c>$ is accepted by the Lock-step Product of M_1 and M_2.

Product
The Product of two automata M_1 and M_2 yields an automaton, that accepts only words, that are accepted by both M_1 and M_2.

$$X = X_1 \cap X_2$$
$$\delta = \{((s_1,s_2),x, (t_1,t_2)) \mid t_1 \in \delta_1(s_1,x) \text{ and } t_2 \in \delta_2(s_2,x) \}.$$

The three mentioned types of products are also valid for M-automata. In that case, the set of limitsets C is given by:

$$C = \{ c \subseteq Q \mid i^{th} \text{ coordinates of } q \in c \text{ form a limitset} \in C_i \text{ for } 1 \le i \le 2 \}$$

4.3.4 Miscellaneous operations

An important function in FADELA is generating a finite automaton that accepts the language defined by a given regular expression. This is done in a very straightforward way:
The tree representing the RE is recursively (depth first) translated into a FA, using the functions for union, concatenation and (ω-)closure. This is in fact the implementation of the construction discussed in paragraph 2.7.

Reflection of an automaton yields an automaton that accepts exactly the reverse (cf. equation 2.2) of the language accepted by the given automaton. Of course this is only defined for finite words. It is implemented by simply reversing the transition-matrix M and swapping I and F:

$$M_{refl}[i][j] = M[j][i]$$
$$I_{refl} = F$$
$$F_{refl} = I$$

The user needs the possibility to define automata, by directly describing them in stead of having FADELA form them from regular expressions. An automaton is defined by assigning a description of it to a variable. The description contains the set of initial states I, final states F and transitions δ:

```
M = {
{ 0 }
{ 1, 2 }
( 0, a, 1 )
( 0, b, 2 )
( 1, b, 2 )
}
```

To make input of these descriptions easier, the syntax is not very strict and the same definition is given by:

```
M = { 0 {1,2}
0 a 1
0 b 2
1 b 2
}
```

A M-automaton is defined the same way, but of course the set of limitsets *C* is given in stead of final stateset *F*:

```
M = { MULLER
{ 0 }
{ { 1, 2 }, { 2 } }
( 0, a, 1 )
( 1, a, 2 )
( 2, b, 1 )
}
```

The indication 'MULLER' should of course not be necessary, since the description of *C* is clearly distinguishable from *F* in 'normal' automata. In the case that *C* or *F* are empty, the difference can not be seen, but then again, such automata are of no real concern, because they only accept the empty language. If B-automata are added to FADELA, a distinction between B-automata and 'normal' ones is needed, which can be done by the indication 'BUCHI'. Now the 'MULLER'-indication is merely an extra indication towards the user.

In order to examine the behaviour of automata, some functions are implemented to retrieve information about (parts of) the automata. These functions are queries for initial, final or limit states, computation of the ε-closure of a set of states and computation of the next or previous states upon a given input(-vector), a given word or simply all possible next and previous states, reachable in one transition.

4.4 Computation of regular expressions

The most important function to enable examination of the behaviour of an automaton is computing a regular expression for the language accepted by the automaton, since regular expressions are usually more easily interpreted than descriptions of the finite automaton itself.

This computation is done in FADELA using the methods described in paragraph 2.8 and 3.5. The construction of RE's for all paths, following equations 2.14 and 2.15 is not done for all paths from I to F, as needed in equation 2.16. In stead, this method is only applied to strong components within Q, i.e. sets of states in which every state can reach any other state. Clearly, these strong components are the only parts where loops exist and thus the only places causing closures in the RE. The method of paragraph 2.8 was needed to cope with these loops (and closures), but in parts, where there are no loops, RE can be formed by concatenating RE's along the paths.

The computation of RE's is now divided in two steps:

For every strong component, the RE is computed using equations 2.14-2.16, taking as initial states those states that are really initial or are reached by states outside the component. Final states and states reaching states not in the component are considered final states. Here reaching means 'having a transition towards'.

The global RE is the union of the RE's along all paths from initial to final states. The RE along a path is the concatenation of symbols on edges between strong components and previously computed RE within strong components. In the case of M-automata, the final RE is computed using equation 3.11.

The method of paragraph 2.8 leads to a lot of redundant information in the final RE. Some simplifications are already mentioned in paragraph 2.8. FADELA offers a general simplification of RE, which attempts to decrease the size of RE's. This is done by recursive (depth first) replacement of subexpressions by equivalent expressions, having fewer terms. It does not transform RE into a standard form, nor does this simplification imply, that a global minimum for the size of RE is reached. The used simplifications are shown in tables 1,2 and 3, showing the simplified results of the operations of (ω-)closure, union and concatenation.

table 1. simplification (ω-)closures

$\alpha =$	$\alpha^\omega =$	$\alpha^* =$
ε	ε	ε
β^*	β^ω	β^*
$\beta+\varepsilon$	β^ω	β^*
$\beta+\beta^*$	β^ω	β^*
$\beta.\beta^*$	β^ω	β^*

table 2. simplification unions

$\alpha =$	$\alpha_{simpl} =$
$\beta.\beta^*+\varepsilon$	β^*
$\beta^*+\varepsilon+\gamma$	$\beta^*+\gamma$
$\beta+\beta$	β
$\beta+\beta^*$	β^*
$\alpha+(\alpha+\beta)^*$	$(\alpha+\beta)^*$

table 3. simplification concatenations

$\alpha =$	$\alpha_{simpl} =$
$\beta.\varepsilon.\gamma$	$\beta.\gamma$
$\beta.\beta^\omega$	β^ω
$\beta^*.\beta^\omega$	β^ω
$\beta^*.\beta^*$	β^*
$(\beta+\varepsilon).\beta^\omega$	β^ω
$(\beta+\varepsilon).\beta^*$	β^*

5. *EXAMPLE OF FADELA*

In this chapter, an example of the use of FADELA is given. A batch performing a test of containment of one language by another is used.

The following batch expects two languages l1 and l2 to be defined. If the text of this batch is in file 'contain', then after defining l1 and l2, the user must give the command

```
# contain
```

which opens file 'contain' for input, thus executing the batch and returning to normal input afterwards. The text of the batch is:

```
m1 = M(l1)
m2 = M(l2)
print ("The language ";l1;" is ";)
on not empty (prod (m1,compl (m2,labs (m1)))) print ("NOT ";)
print ("contained in the language ";l2)
```

First two automata m1 and m2 accepting exactly the languages l1 and l2 respectively are constructed. The boolean expression controlling the ON-statement does the following:

— m2 is complemented with respect to all symbols of m1 and m2, yielding an automaton accepting all possible words over input symbols of m1 and m2, except those words accepted by m2.

— The product of m1 and the complement of m2 is constructed, yielding an automaton accepting only words accepted by m1 but not accepted by m2.

— If this product does accept any words, then these clearly are words that are in l1 but not in l2. This implies, that l1 is NOT completely contained in l2 and therefore the result of the expression is FALSE.

A run of FADELA (reply from FADELA is shown bold) now may look like:

```
l1 = a.b*.(a+b+c)W
l2 = a.bW
#contain
Warning: Minimal Muller
Warning: Minimal Muller
Warning: Minimal Muller
The language a.b*.(a+b+c)W is NOT contained in the language a.(b)W
l3 = l1
l1 = l2
l2 = l3
#contain
Warning: Minimal Muller
Warning: Minimal Muller
Warning: Minimal Muller
The language a.(b)W is contained in the language a.b*.(a+b+c)W
```

The warnings are a result of the fact that in FADELA the automata are minimized before taking the product or complement, but no minimization is implemented for M-automata, in which case only determinization is applied. The capital W represents $^{\omega}$.

6. CONCLUSIONS AND RECOMMENDATIONS

The most important operations on automata and (ω-)regular expressions are available in FADELA.

Constructing an equivalent deterministic automaton can be done, not only for nondeterministic 'normal' automata, but also for M-automata. The possibility was found in literature, where proof for the equivalence between deterministic and non-deterministic M-automata is discussed, but that proof involved constructing an ω-regular expression for the language accepted by the non-deterministic M-automaton and proving that for any ω-regular expression an accepting deterministic M-automaton can be found. The construction discussed in this report is a direct conversion from non-deterministic to deterministic automata, based on the subset-construction.

Currently, only M-automata accepting ω-regular languages are available in FADELA, but it should not be very difficult to include other types of automata, such as B-automata.

Minimization of the number of states in an automaton is only implemented for 'normal' automata. Minimization for M-automata, although maybe not global, should be possible too.

7. REFERENCES

[1] *ALPERN, Bowen and Fred B. SCHNEIDER*
Recognizing safety and liveness.
Distributed Computing, vol 2 (1987), pp. 117-126.

[2] *AGGARWAL, S., C. COURCOUBETIS and P. WOLPER*
Adding liveness properties to coupled finite-state machines.
ACM Transactions on Programming Languages and Systems, vol 12 (1990), pp. 303-339.

[3] *TRAKHTENBROT, B.A. and Ya.M. BARZDIN'*
Finite automata.
Vol. 1: Behavior and Synthesis.
Translated from the Russian.
Amsterdam: North-Holland, 1973.

[4] *EILENBERG, Samuel*
Automata, languages, and machines.
Vol. A.
New York: Academic Press, 1974.

[5] *PEREMANS, W*
Automatentheorie en formele talen.
Faculteit Wiskunde en Informatica, Technische Universiteit Eindhoven, 1985-1987.
Collegedictaatnummer 2337.

[6] *McNAUGHTON, R. and H. YAMADA*
Regular expressions and state graphs for automata.
IRE Transactions on Electronic Computers, vol 9 (1960), pp. 39-47.

[7] *HOPCROFT, J.E. and J.D. ULLMAN*
Introduction to automata theory, languages and computation.
Addison-Wesley, 1979.

[8] *ALAIWAN, H*
Equivalence of infinite behavior of finite automata.
Theoretical Computer Science, vol 31 (1984), pp. 297-306.

[9] *McNAUGHTON, Robert*
Testing and generating infinite sequences by a finite automaton.
Information and Control, vol 9 (1966), pp. 521-530.

[10] *MORIYA, Tetsuo and Hideki YAMASAKI*
Accepting conditions for automata on ω-languages.
Theoretical Computer Science, vol 61 (1988), pp. 137-147.

[11] *CHOUEKA, Yaacov*
Theories of automata on ω-tapes: a simplified approach.
Journal of Computer and System Sciences, vol 8 (1974), pp. 117-141

[12] *BÜCHI, J.R.*
On a decision method in restricted second order arithmetic.
Proc. Int. Congr. on Logic, Methodology and Philosophy of Science. Stanford, California, (1960), pp. 1-11.
Stanford University Press, 1962.

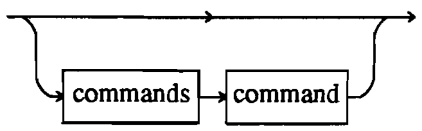
[13] *MULLER, D.E.*
Infinite sequences and finite machines.
Proc. 4th Ann. Symp. on Switching Circuit Theory and Logical Design. Chicago (1963), pp. 3-16.
Inst. of Electrical and Electronic Engineers, New York.

[14] *HOPCROFT, J.E.*
An n log n algorithm for minimizing the states in a finite automaton.
in Theory of Machines and Computations, pp. 189-196.
Academic Press, 1971.

[15] *GRIES, David*
Hopcroft's algorithm.
Acta Informatica, vol 2 (1973), pp. 97-109.

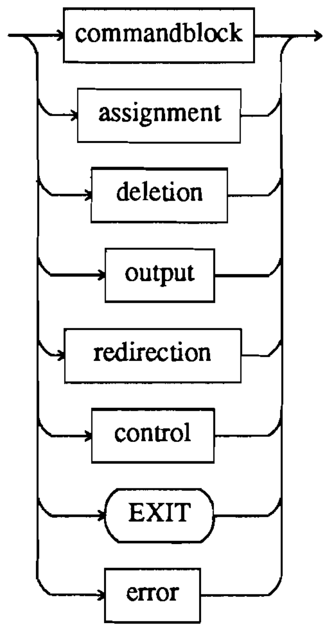
APPENDIX A: FADELA SYNTAX AND SEMANTICS

commands:



Commands are given by the user interactively or read from a batch file. Once a complete command is read (possibly containing subcommands in the case of control commands), its execution starts and the parsetree for it is deleted.

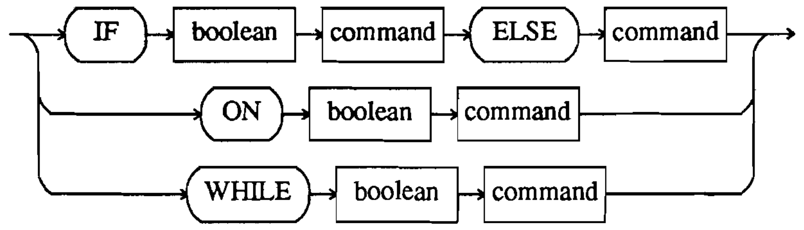
command:



EXIT immediately stops FADELA, closing all opened files.
At any level in the parsetree a syntax error may occur. If so, the execution of the command at the highest level will not take place and an error message with an indication of the error is given. This error looks like:

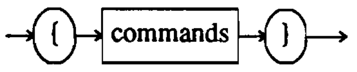
syntax error in line <nn>: '<symbol>' not expected.

control:



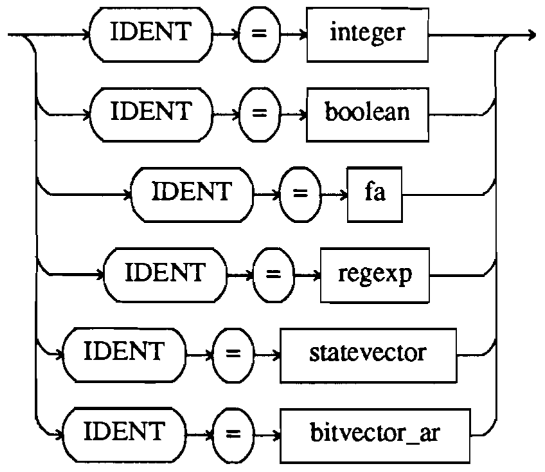
The boolean expression is evaluated and depending on the result, the first (result = TRUE) or second (result = FALSE) subcommand of the IF-command is executed.
The ON-command is equivalent to an IF-command, where the second subcommand is empty.
The WHILE-command is equivalent to an ON-command, that is repeated, as long as the result of the boolean expression is TRUE.

commandblock:



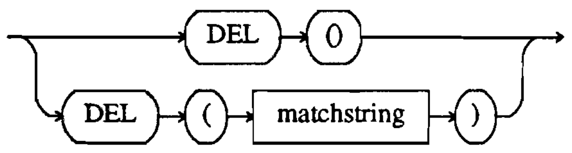
Several commands are grouped, to form one (sub-)command.

assignment:



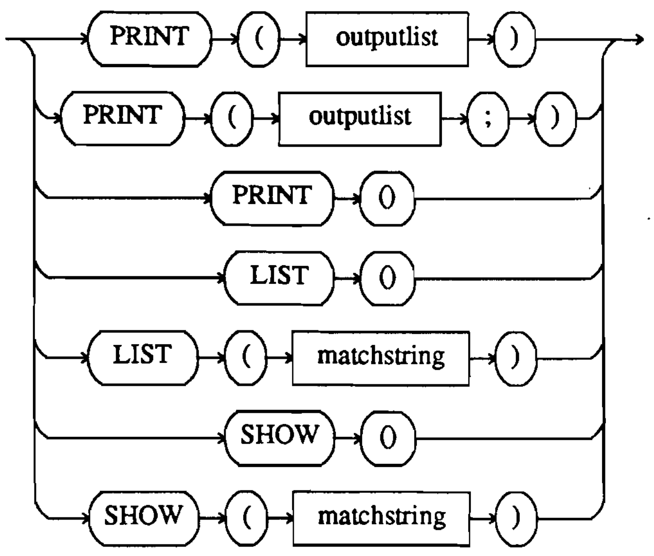
An identifier represents a variable and consists of elements of
{ A,..,V,X,..,Z,a,..,z,0,..,9,_,',@ }, starting with an element of { A,..,V,X,..,Z,a,..,z }. The 'W' can not be used, because it is used as a special symbol, namely ω in regular expressions. To the variable, the result of the right-hand side expression is assigned. Once a value has been assigned, the variables type (integer, boolean, fa, re, set of states or word) is set and any further assignments to it must have the same type. If the right-hand side expression does not have the correct type, a syntax error results.

deletion:



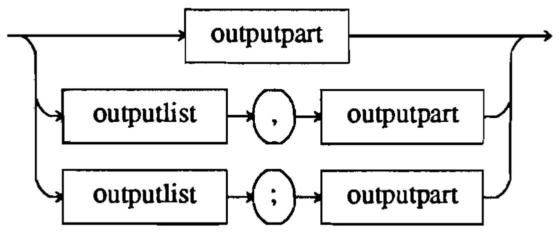
<matchstring> may be any string of characters. All variables that have names starting with <matchstring> are deleted. If no <matchstring> is given, all variables are deleted. After deletion, the variable-names are free for new assignments of any type.

output:

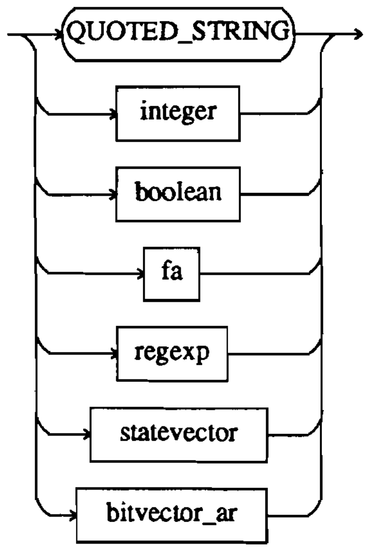


PRINT sends all items in <outputlist> to the current output, adding a newline after each item, unless it is followed by ';'.
LIST sends to the current output all variable-names starting with <matchstring> or all variable-names if no <matchstring> is given.
SHOW does the same as list, but not only shows the names, but the values as well.

outputlist:

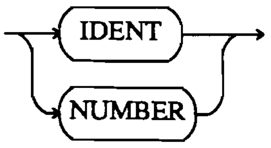


outputpart:

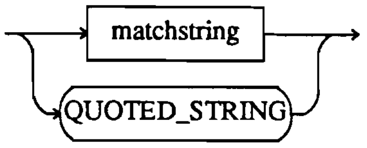


An outputpart is an item printed by PRINT(outputlist). These may be strings, that must be given in double quotes '"' or expressions.

matchstring:

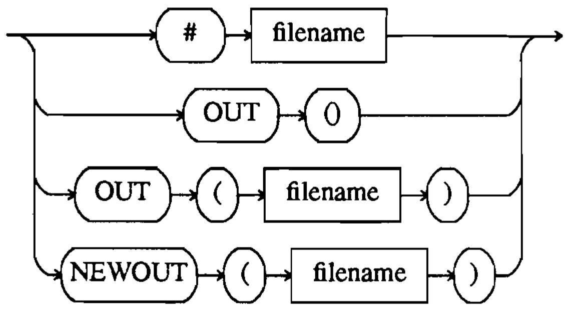


filename:



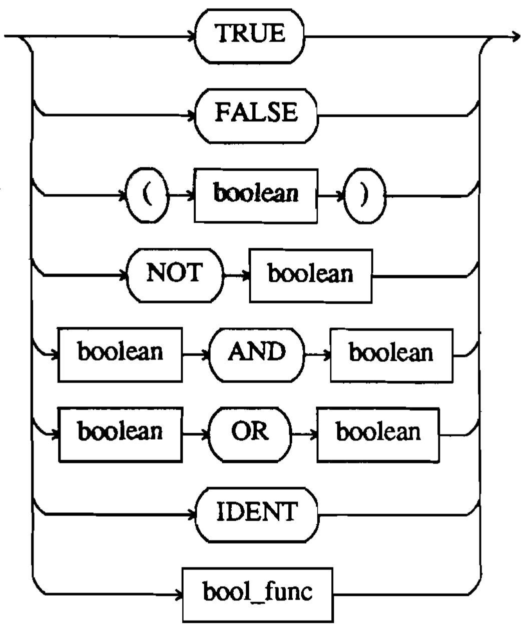
If a file is not in the current directory, it must be referred to by giving its path within double quotes '"'.

redirection:

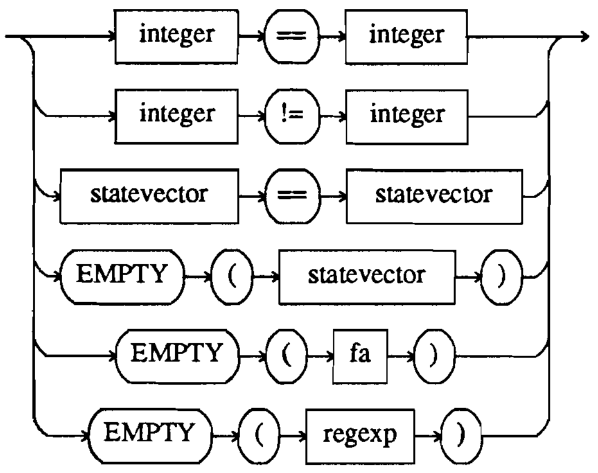


Input can be redirected to come from a (batch-)file, by using the '#' inclusion command. After this command, commands will be read from the given file, until the end of that file is reached. Then that file is closed and further commands are taken from the current input-stream. File inclusions may be nested, but the depth is restricted by the number of files that may be open at the same time. If the input-file is not found, an error is given:
ERROR: cannot open '<filename>' for input
Output can be redirected to a file, by giving the OUT or NEWOUT command.
OUT() closes the current output file and further output will go to standard out.
OUT(filename) closes the current output and opens the file <filename> for further output, appending to it, if it already exists.
NEWOUT(filename) does the same, but always opens the file overwriting existing files of that name.

boolean:

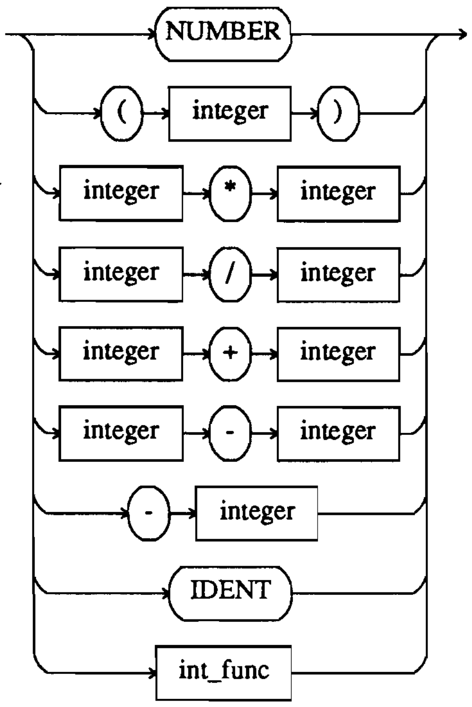


bool_func:

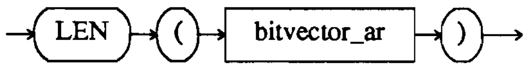


EMPTY has value TRUE, if and only if the argument (set of states, fa or re) is empty, which in the case of fa means, that fa does not accept any word. Boolean expressions are evaluated following the usual precedence: NOT, AND , OR.

integer:

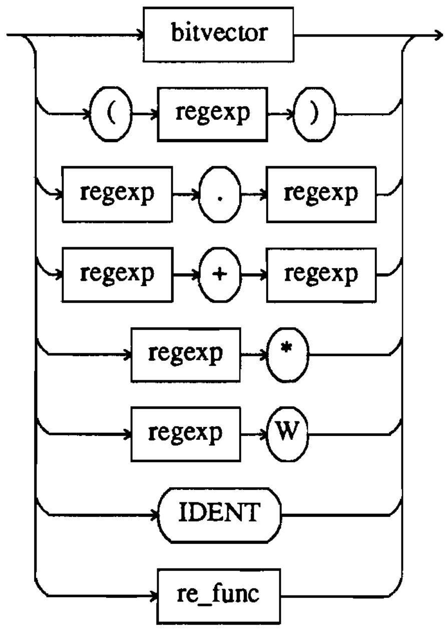


int_func:

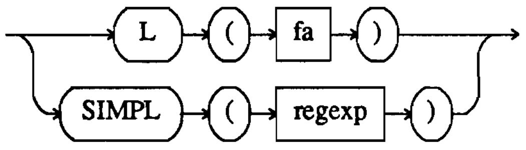


LEN returns the number of symbols (i.e. boolean products, bitvectors) that are in the given word (i.e. bitvector array).

regexp:

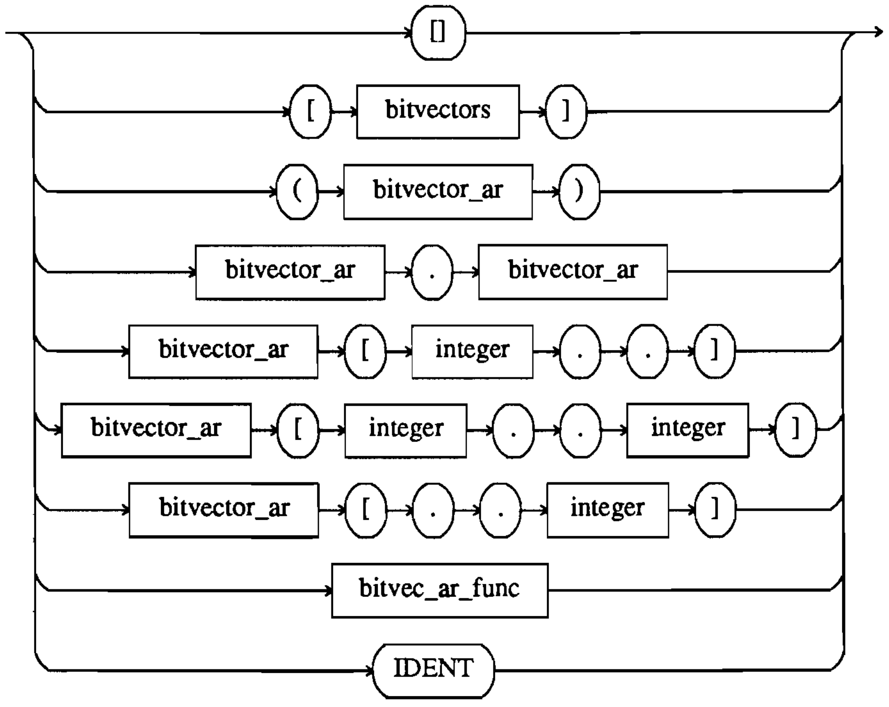


re_func:

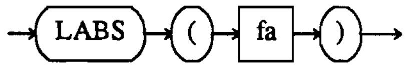


Regular expressions have the usual form. 'W' stands for ω in ω-regular expressions.
L(fa) returns a (simplified) (ω-)regular expression for the language accepted by <fa>.
This may give errors, if unions, closures or concatenations occur, that are not possible, because of incompatible arguments. These errors are listed under fa_func.
SIMPL(re) returns a simplified copy of <re>.

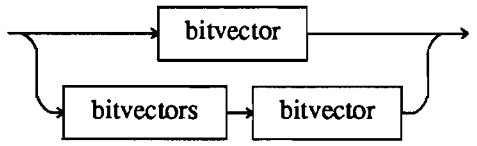
bitvector_ar:



bitvec_ar_func:



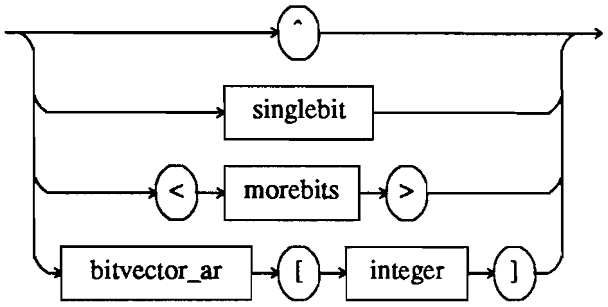
bitvectors:



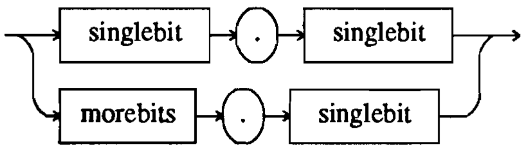
LABS(fa) returns a word (bitvector array) consisting of all input symbols of <fa>.

w1.w2 yields the concatenation of w1 and w2.
w[i..j] is the slice of w from position i to j, where w = w[1].w[2].w[3]....w[LEN(w)].
If j < i then the result is w[i].w[i-1].w[i-2]....w[j+1].w[j].
w[..j] is equivalent to w[1..j].
w[i..] is equivalent to w[i..LEN(w)].

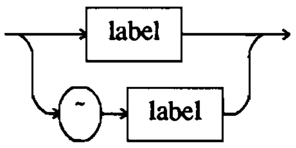
bitvector:



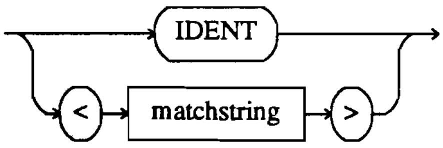
morebits:



singlebit:

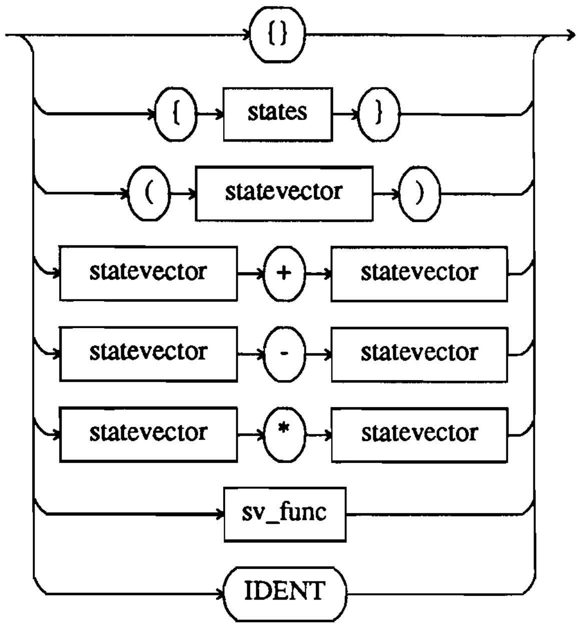


label:

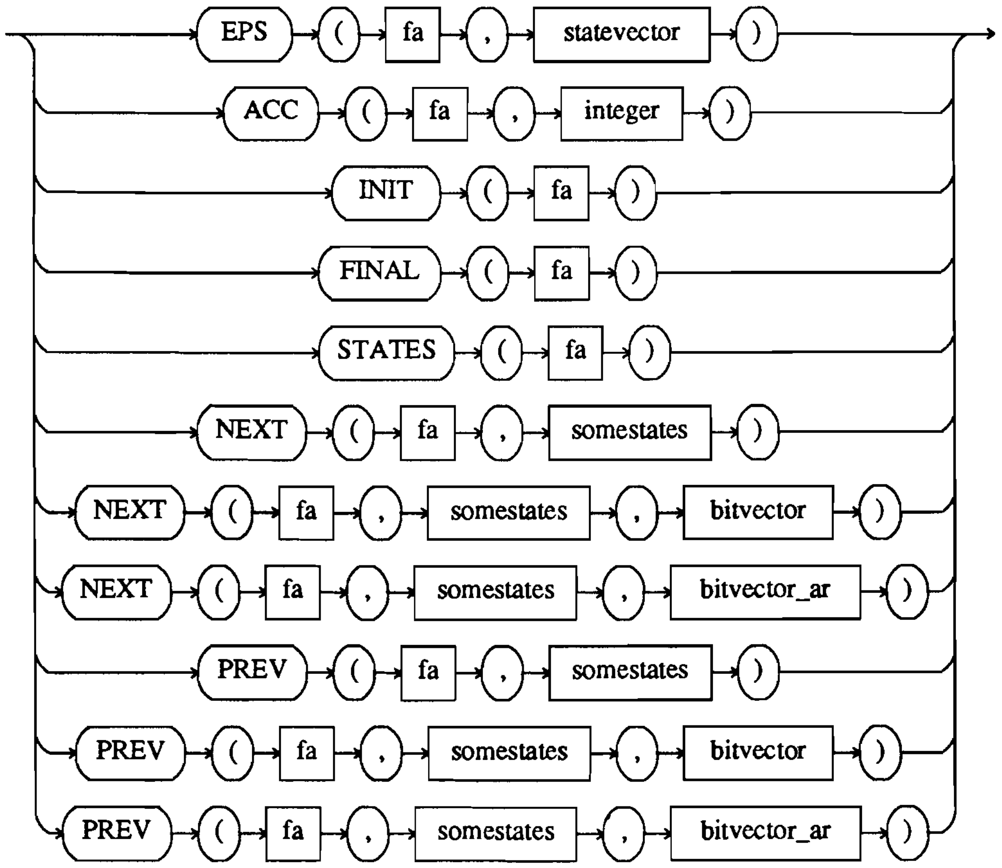


In boolean functions (products, bitvectors) '^' represents ε. '~' is the negation of the following bit. Bits (atomic booleans) may have any name, but sometimes the name must be given between '<' and '>' to prevent syntax errors, when the name has been used for a variable.
A bitvector is represented by bits, separated by ',' and between '<' and '>'.

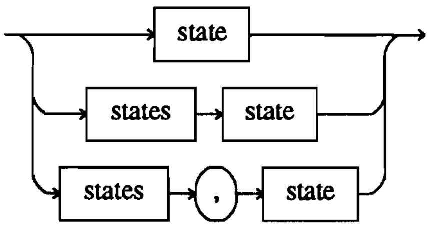
statevector:



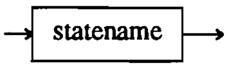
sv_func:


EPS (fa , statevector)
ACC (fa , integer)
INIT (fa)
FINAL (fa)
STATES (fa)
NEXT (fa , somestates)
NEXT (fa , somestates , bitvector)
NEXT (fa , somestates , bitvector_ar)
PREV (fa , somestates)
PREV (fa , somestates , bitvector)
PREV (fa , somestates , bitvector_ar)

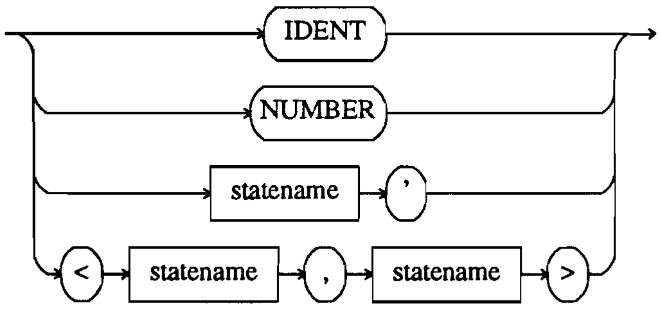
states:



state:



statename:

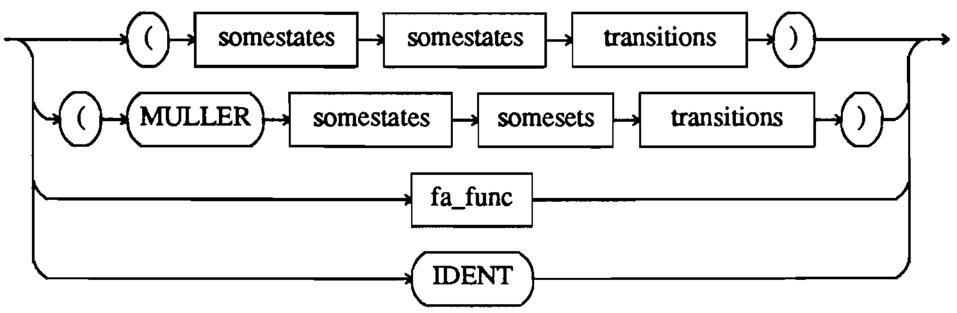


On sets of states (statevectors) the functions of union (+), difference (-) and intersection (*) are allowed.
States are represented by identifiers or numbers or by pairs of statenames, which result from product operations on automata. In these pairs, two statenames, separated by ‘,’ are within ‘<’ and ‘>’. Functions returning sets of states are:

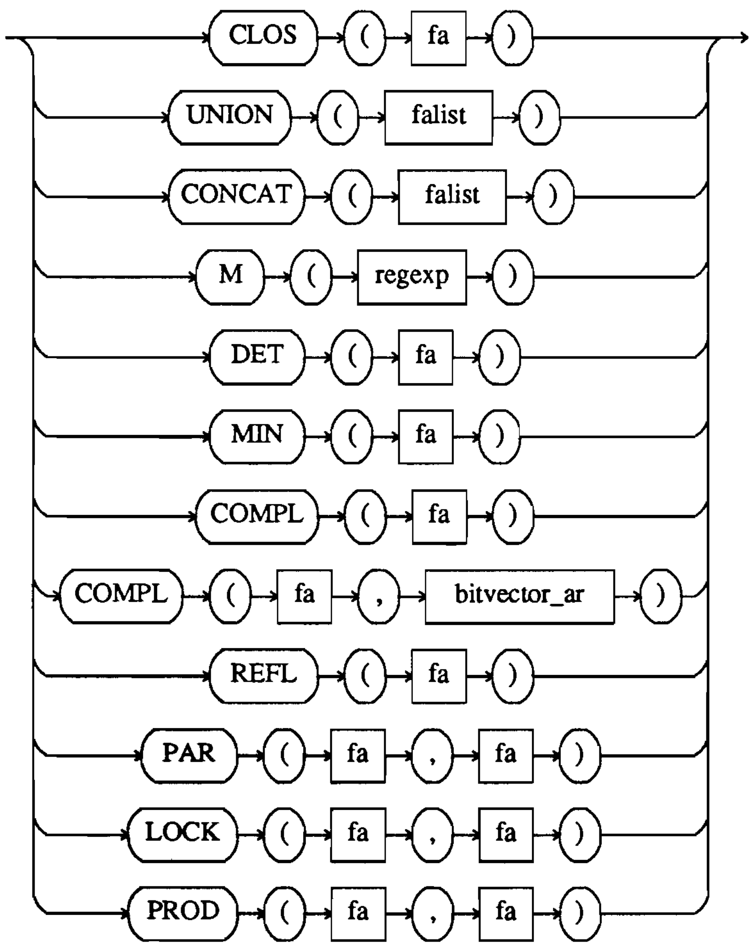
— EPS(fa,set) returning the ε-closure of <set>.
— INIT(fa) returning the initial states of <fa>.
— FINAL(fa) returning the final states of <fa>.
— ACC(fa,n) returning the <n>-th limitset of <fa>, empty if <fa> has less than <n> limitsets.
— STATES(fa) returning the set of all states of <fa>.
— NEXT(fa,set) returning all states reachable from states in <set> on any input symbol.
— PREV(fa,set) returning all states that reach states in <set> on any input symbol.
— NEXT(fa,set,x) returning all states reached from <set> on input symbol <x>.
— PREV(fa,set,x) returning all states reaching <set> on input symbol <x>.
— NEXT(fa,set,w) returning all states reached from <set> on input of word <w>.
— PREV(fa,set,w) returning all states reaching <set> on input of the reverse of word <w>.

In the functions NEXT and PREV, a singleton set may be given by the element without '{' and '}'.

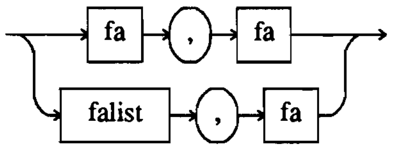
fa:



fa_func:



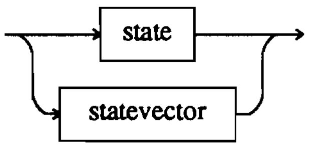
falist:



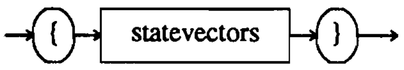
Functions returning FA's are:

- CLOS(fa) returns a FA accepting the language (L(fa))*. If <fa> is a M-automaton, then this function is not defined and an empty FA is returned. In that case also an error-message is given:
  ERROR: Closure of a Muller-FA
- CONCAT(fa1,fa2,..,fan) returns a FA accepting the concatenation of the languages accepted by <fa1>,<fa2>,.. and <fan>. If any one of the arguments except <fan> is a M-automaton, then an empty FA is returned and an error-message given:
  ERROR: Concat Muller-FA
- UNION(fa1,fa2,..,fan) returns a FA accepting the union of the languages accepted by the arguments. Those arguments must either all be M-automata or all 'normal' automata, else an empty FA is returned and an error-message given:
  ERROR: Union of a Muller-FA and a NotMuller-FA
- M(re) returning an automaton accepting exactly the language described by <re>. It uses the functions of union, concatenation and (ω)-closure and thus may give the error-messages produced to those functions.
- DET(fa) returning a deterministic equivalent of <fa>, using the subset construction.
- MIN(fa) returning the minimal equivalent DFA of <fa>, giving a warning if <fa> is a M-automaton. The result then is a correct DFA, though not minimal and the message is:
  Warning: Minimal Muller.
- COMPL(fa,word) returning the complementary FA of <fa>, i.e. a FA that accepts all words not accepted by <fa>. If the second argument is omitted, then the resulting automaton accepts only words over the input-alphabet of <fa>, else all symbols in <word> (bitvector array) are added to this alphabet, before constructing the complement.
- REFL(fa) returning a FA accepting the reverse of the language accepted by <fa>. If <fa> is a M-automaton, then this function returns an empty FA and gives an error-message:
  ERROR: Reflection Muller-FA
- PAR(fa1,fa2) returning the product of <fa1> and <fa2>, described in paragraph 4.3.3 under parallelization. The arguments must either be both M-automata or 'normal' automata, else the following error-message is given:
  ERROR: Parallel of a Muller-FA and a NotMuller-FA
- LOCK(fa1,fa2) returning the product of <fa1> and <fa2>, described under Lock-step product in paragraph 4.3.3. Possible error:
  ERROR: LockstepProduct of a Muller-FA and a NotMuller-FA
- PROD(fa1,fa2) returning the product of <fa1> and <fa2>, accepting only words that are accepted by both <fa1> and <fa2>. Possible error:
  ERROR: Product of a Muller-FA and a NotMuller-FA

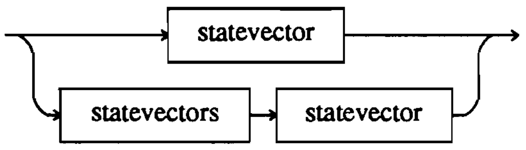
somestates:



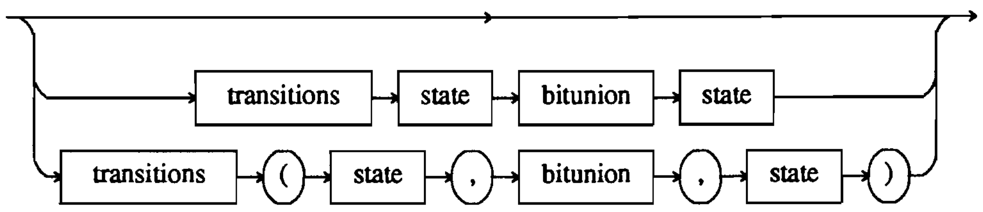
somesets:



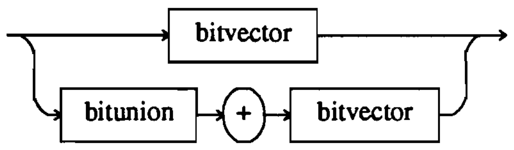
statevectors:



transitions:



bitunion:


bitvector
bitunion
+
bitvector