An efficient algorithm which determines the output from a sequential machine for regular inputs*

Kenneth B. Salomon

Department of Mathematics, California State University at Hayward, Hayward, California, 94542, USA

This paper deals with the following problem from automata theory: given a deterministic sequential machine, calculate its set of outputs when a given regular set is used as input. The output is known to be regular and various effective methods have been developed in the past to determine it. However these methods tend to be quite inefficient and ill-suited to either hand or computer calculation. The procedure developed here is sufficiently efficient and straightforward as to be suitable for machine implementation and, in fact, makes use of an already-existing SNOBOL IV program.

(Received April 1973)

Downloaded from https://academic.oup.com/comjnl/article/17/4/349/443732 by guest on 19 April 2024

1. Introduction

The translation of regular expressions by finite-state deterministic sequential machines has been studied by many authors (e.g. Ginsburg and Rose, 1963; Ginsburg and Hibbard, 1964; Ginsburg and Spanier, 1966; Ullian, 1967) as a mathematical model of certain aspects of compiling. In particular, the result that a given type of sequential machine preserves regular sets has been proven for various types of machines. In the proof of these results a procedure (usually effective, though not always) is presented which converts an input regular expression into the appropriate output regular expression. Even those which are effective are inefficient for computer implementation and definitely unmanageable by hand.

In this paper an effective procedure is presented which will determine the output from either a deterministic complete sequential machine or a deterministic generalised sequential machine in an efficient manner. The procedure makes use of an algorithm developed by Smith and Yau (1972) for finding the regular expression defined by a finite automaton. As their algorithm has been programmed in SNOBOL IV, the method developed in this paper is clearly available for computer implementation.

The medium for expressing the method will be a deterministic generalised sequential machine with accepting states. This device will be defined and some of its relevant properties developed in the sequel.

2. The a-GSM model

The model which will be used to formulate the procedure is a hybrid of the following well-known devices: a deterministic generalised sequential machine (GSM, for short) and a deterministic finite-state acceptor (FSA, for short). The latter is also called a finite automaton by many authors. The reader is referred to the standard literature (e.g. Ginsburg, 1966) for further development of the properties of these devices. In an appendix to this paper we review the notation and definitions of regular sets and regular expressions used in the sequel.

Definition 1. A *deterministic generalised sequential machine with accepting states* (a-GSM, for short) is a 7-tuple

$$M = (Q, \Sigma, \Delta, f, g, q_0, F) \ ,$$

where

Q is a finite set of states of M.
Σ is a finite set of input symbols to M.
Δ is a finite set of output symbols from M.
$f: Q \times \Sigma \rightarrow Q$ is the next-state function of M.
$g: Q \times \Sigma \rightarrow \Delta^*$ is the output function of M.
$q_0 \in Q$ is the initial state of M.
$F \subseteq Q$ is the set of accepting states of M.

The states in F will be denoted by double circles in graphical representations of a-GSM's. The functions f and g are extended to $Q \times \Sigma^*$ in the usual way. We now describe how the device operates.

Definition 2

Let $M = (Q, \Sigma, \Delta, f, g, q_0, F)$ be an a-GSM. For each input sequence $w \in \Sigma^*$, and each $q \in Q$, let

$$M_p(q, w) = g(q, w)$$

be called the *partial output of M* when started in state q with input w. $M_p(q, w)$ will be called the *acceptable output*, or simply the *output* if no confusion will result, if and only if $f(q, w) \in F$. For sets of input sequences $S \subseteq \Sigma^*$ we will define

$$M_p(q, S) = \{M_p(q, w) \mid w \in S\}$$

and

$$M(q, S) = \{M_p(q, w) \mid w \in S \text{ and } f(q, w) \in F\}$$

Whenever $q = q_0$ we will usually write simply $M_p(w)$, $M(w)$, $M_p(S)$ and $M(S)$ for $M_p(q_0, w)$, $M(q_0, w)$, $M_p(q_0, S)$ and $M(q_0, S)$, respectively. The operation $M(S)$ defined by an a-GSM, M, processing a set of inputs, S, will be called an *a-GSM translation*.

Note that the action of an a-GSM can be likened to a buffer which must be filled from a certain set before its contents can be used. The interpretation of an a-GSM as a hybrid GSM-FSA should now be clear. We formalise it in the following way.

Theorem 1:

Let $M = (Q, \Sigma, \Delta, f, g, q_0, F)$ be an a-GSM and R be any regular set. Then $M(R)$ is regular.

Proof:

Considering M first simply as an FSA and ignoring its output we see that the set of inputs, $R_1 \subseteq \Sigma^*$, which are accepted, i.e. which drive M into any of the states in F, is regular. Now taking $R_2 = R_1 \cap R$, which is regular, we see that

$$\begin{aligned} M(R) &= \{M_p(y) \mid y \in R \text{ and } f(q_0, y) \in F\} \\ &\{M_p(y) \mid y \in R_2\} \\ &= M_p(R_2) \ . \end{aligned}$$

But $M_p(R_2)$ is, considering M now as a GSM, i.e. taking all states as accepting, just a GSM translation of a regular set. Hence by Theorem 3.3.2 of Ginsburg (1966), which asserts that GSM translations preserve regular sets, we have the result. Q.E.D.

3. Use of the a-GSM as a generator of regular sets

We are now in a position to show that given an arbitrary non-empty regular expression R, one can effectively design an a-GSM M, such that $M(\Sigma_2^*) = R$, where we use the notation

*Some of the results reported in this paper were developed when the author was attending the Cambridge University Summer Logic School under a NATO study grant.

$\Sigma_2 = \{0, 1\}$. It is interesting to note that in this sense a-GSM's are more powerful than GSM's since Ginsburg (1966), Problem 6, p. 102, indicates that no GSM exists which translate Σ_2^* to $R = (a + b)^*c$.

Theorem 2:
Let R be any non-empty regular expression over the alphabet Δ. One can construct an a-GSM $M = (Q, \Sigma_2, \Delta, f, g, q_0, F)$ such that $M(\Sigma_2^*) = R$.

Proof:
We proceed by giving an inductive construction.
Basis: If $R = x$ where $x \in \Delta^*$, then the 2-state a-GSM of **Fig. 1**(*a*) satisfies $M(\Sigma_2^*) = R$. Note: q_1 will be denoted a 'null' state.

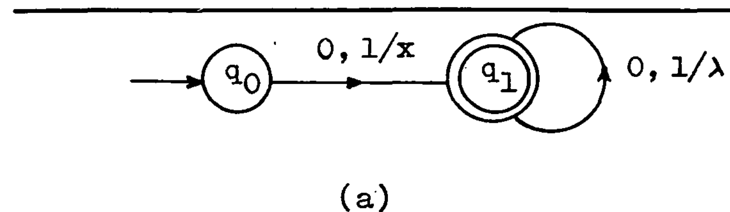
Induction step: Assume that R_1 and R_2 are any two non-empty regular expressions over Δ and that M_1 and M_2 are a-GSM's such that $M_1(\Sigma_2^*) = R_1$ and $M_2(\Sigma_2^*) = R_2$, then:

(i) the a-GSM of Fig. 1(*b*) satisfies $M(\Sigma_2^*) = R_1^*$. (N.B. In Figs. 1(*b*), 1(*c*) and 1(*d*) any branch directed into a box is obtained by breaking the branch loop associated with the null state(s) within the box and redirecting it (them) as shown). Note: q_1 will be denoted a 'null' state.

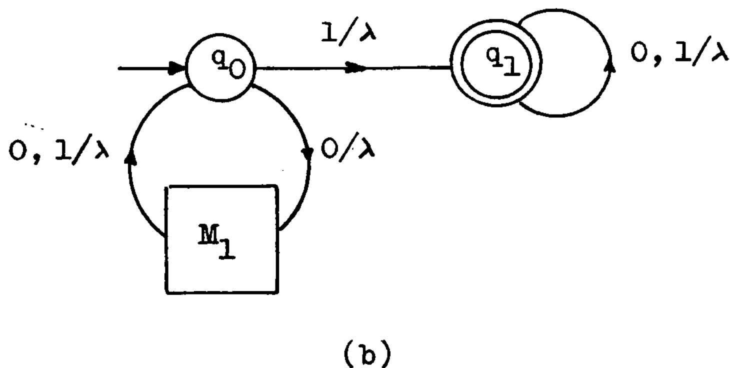
(ii) the a-GSM of Fig. 1(*c*) satisfies $M(\Sigma_2^*) = R_1 + R_2$.

(iii) the a-GSM of Fig. 1(*d*) satisfies $M(\Sigma_2^*) = R_1R_2$. (N.B. In Fig. 1(*d*) the accepting states of M_1 are retained, i.e. not returned to single circles, if and only if $\lambda \in R_2$, which is decidable).

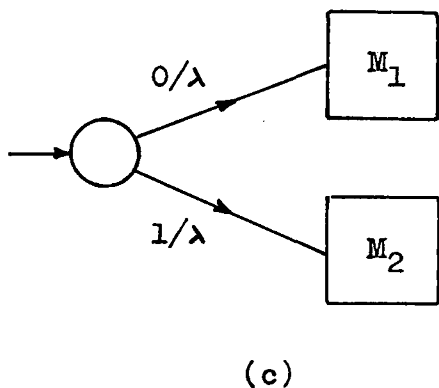
Q.E.D.



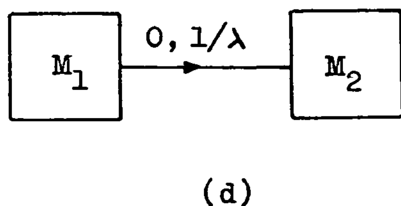
(a)



(b)



(c)



(d)

Fig. 1 Constructions for Theorem 2

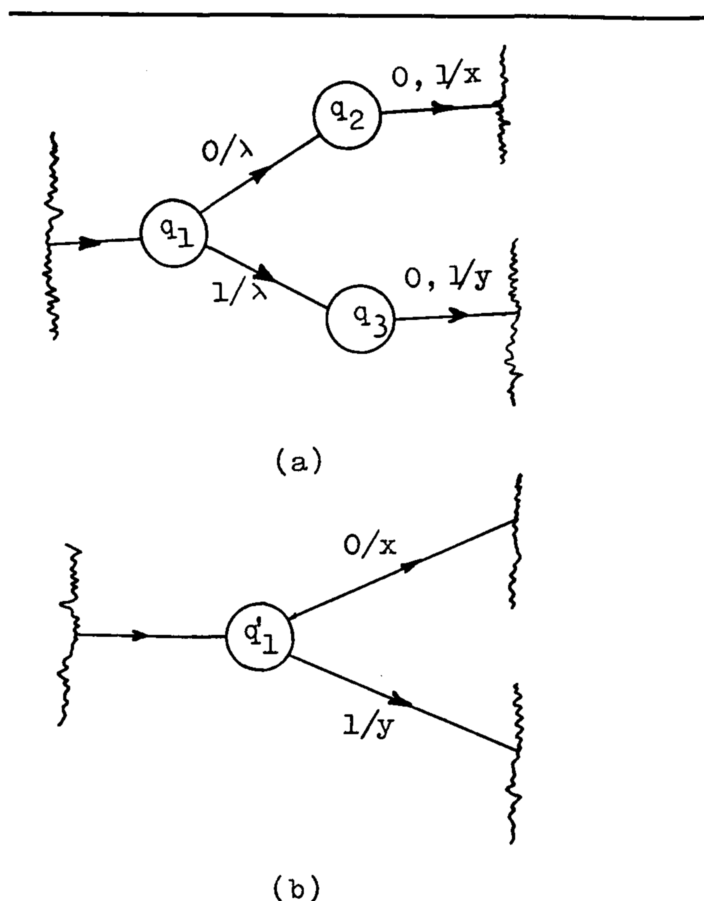


Fig. 2 Rule 2. $x, y \in \Delta^*$

While it is not strictly necessary in obtaining the results of this paper to minimise a-GSM's, some obvious rules will be given to reduce the size of the machines used in the examples. We give three rules which may be applied to the a-GSM constructed in the preceding theorem without changing the regular expression it outputs.

Rule 1
Any two states joined by a branch labelled 0, $1/\lambda$ may be merged into one state. If either of the states is in F, then the resulting state is also. The branch which connected the states is deleted.

Rule 2
When the situation of **Fig. 2**(*a*) obtains, it may be replaced by that shown in Fig. 2(*b*). If $q_1 \in F$, then the resulting state q_1' is also.

Rule 3
When the situation of **Fig. 3**(*a*) obtains, it may be replaced by that shown in Fig. 3(*b*), providing that if $x \neq \lambda$, then $q_2 \notin F$. If $q_1 \in F$, then so is q_1'.

Rule 1 follows since this situation occurs only when two sub-machines have been joined, as can be seen by reviewing the proof of Theorem 2, or a star has been implemented via step (*i*).

Downloaded from https://academic.oup.com/comjnl/article/17/4/349/443732 by guest on 19 April 2024

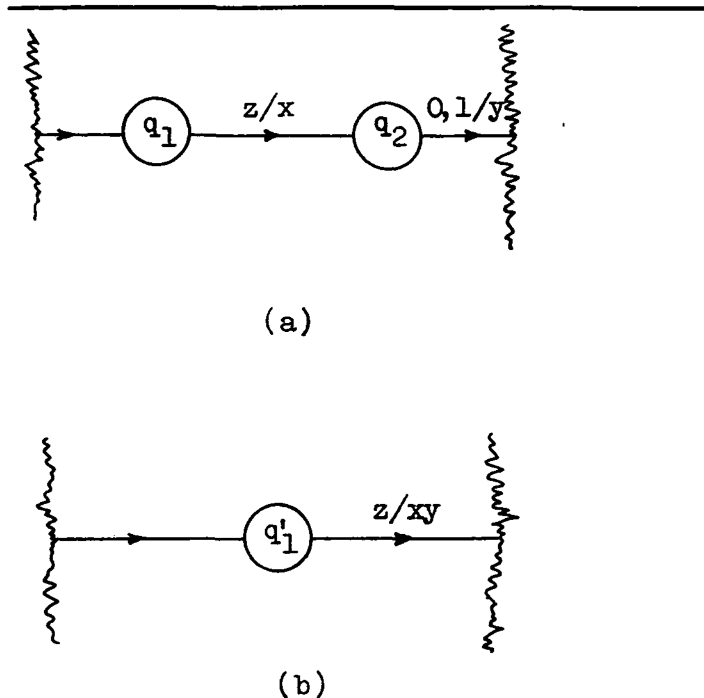


Fig. 3 Rule 3. $x, y \in \Delta^*$, $z \in \{0, 1\}$ or $z = \{0, 1\}$

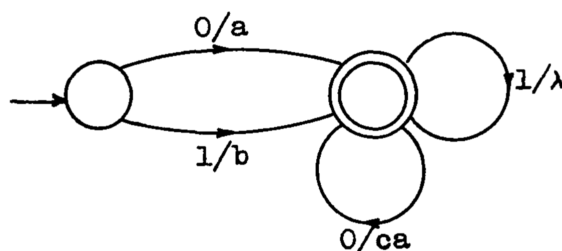


Fig. 4 An a-GSM which outputs $(a + b)(ca)^*$

In either case it is seen that no output sequence has been changed by the merger, since under any input the machine merely changes state and outputs the null sequence. For Rules 2 and 3 it is clear that no output sequence is changed.

The procedure indicated by Theorem 2 (including state merging) applied to

$$R = (a + b)(ca)^*$$

results in the a-GSM of **Fig. 4.**

4. An application of the a-GSM

To describe the procedure for determining the output from a GSM when it translates a regular expression it will be convenient to define the composition of two a-GSM's. This corresponds to their cascade connection, i.e. the output of the first is used as input by the second.

Definition 3:
Let $M_1 = (Q_1, \Sigma, \Delta, f_1, g_1, q_{0_1}, F_1)$ and $M_2 = (Q_2, \Gamma, \Omega, f_2, g_2, q_{0_2}, F_2)$ be a-GSM's such that $\Delta \subseteq \Gamma$. The *composite* of M_1 and M_2, written $M_1 \circ M_2$, is defined to be the a-GSM

$$M_1 \circ M_2 = (Q_1 \times Q_2, \Sigma, \Omega, f_c, g_c, (q_{0_1}, q_{0_2}), F_1 \times F_2) ,$$

where for every $(q_1, q_2) \in Q_1 \times Q_2$ and every $a \in \Sigma$

$$f_c((q_1, q_2), a) = (f_1(q_1, a), f_2(q_2, g_1(q_1, a))) ,$$
$$g_c((q_1, q_2), a) = g_2(q_2, g_1(q_1, a)) .$$

It should be noted that $M_1 \circ M_2$ *is* an a-GSM and that it possesses the property that for any $R \subseteq \Sigma^*$,

$$M_1 \circ M_2(R) = M_2(M_1(R)) .$$

As the procedure to be given in this section is an adaptation of the algorithm of Smith and Yau (1972) which determines the regular expression defined by an FSA, and as their method uses the concepts of the *derivative* and *integral* of a regular expression, the appendix briefly outlines these terms. For a more thorough treatment consult their paper.

We now review some terms employed in the algorithm to be presented shortly. The first involves the concept of associating derivatives with states of an FSA. (Again this is more fully developed in Smith and Yau). Letting M be an FSA and R be a regular expression denoting the regular set it accepts, it can be shown that each state, q_j, in M can be represented by a regular expression which is a derivative of R with respect to some input word w in the sense that M started in q_j accepts D_wR. One then calls D_wR a *derivative for state* q_j. Thus one can formulate the following.

Definition 4:
A derivative D_wR for state q_j is *minimal* if and only if there exists no other derivative D_xR for state q_j such that x is less lexicographically than w. In this case we shall occasionally write $D_wR = q_j$, where q_j is understood to represent both the state and the regular expression representing the state.

Thus given an FSA one can effectively find a unique list of minimal derivatives to represent its states. This allows one to form a *tree* from the list of minimal derivatives in the following manner: starting from the minimal derivative for the start state, $D_\lambda R$, draw a branch to each $D_{a_i}R$, where a_i is an input symbol. If $D_{a_i}R$ is not minimal, then no branches need be drawn from it. If $D_{a_i}R$ is minimal, draw a branch from $D_{a_i}R$ to each $D_{a_ia_j}R$, where a_j is an input symbol. Apply this process repeatedly until the list of minimal derivatives is exhausted. The corresponding state is next associated with each of these derivatives. (If the FSA contains any null states and the derivative is associated with a null state, the corresponding state is replaced by $\emptyset$).

Finally, as a convenience to the reader we summarise four rules for manipulating integrals developed by Smith and Yau (1972) which are used in the following algorithm.

Rule 1:
Combinatory rule. If we have the integrals over the input alphabet $\{a_i, a_j\}$

$$\int D_{wa_i}R \, da_i = a_iS_1$$

and

$$\int D_{wa_j}R \, da_j = a_jS_2 ;$$

then we have the integral $D_wR = a_iS_1 + a_jS_2$.

Rule 2:
Output rule. If the state represented by this integral is an accept state, add λ to the integral.

Rule 3:
Substitution rule. If state q_k is associated with the regular expression S_1 and the state q_j with the regular expression S_2q_k, then q_j corresponds to S_2S_1.

Rule 4:
Star rule. If a state q_k is associated with $S_1q_k + S_2 + S_3$, then the integral of q_k corresponds to $S_1^*(S_2 + S_3)$.

We are now in a position to present the procedure promised earlier.

Downloaded from https://academic.oup.com/comjnl/article/17/4/349/443732 by guest on 19 April 2024

Algorithm 1:
Let R be any non-empty regular expression over a subset of Δ and $M = (Q_1, \Delta, \Omega, f_1, g_1, q_{0_1}, F_1)$ be any a-GSM. Then the following steps determine a regular expression for $M(R)$:

1. Apply the procedure of Theorem 2 to R producing the a-GSM $N = (Q_2, \Sigma_2, \Delta, f_2, g_2, q_{0_2}, F_2)$ satisfying $N(\Sigma_2^*) = R$.
2. Construct the composition a-GSM $N \circ M = (T, \Sigma_2, \Omega, f_c, g_c, t_0, F_2 \times F_1)$.
3. Considering $N \circ M$ as an FSA (i.e. ignoring outputs) form the lexicographical list of minimal input derivatives beginning with $D_\lambda R_1$, where R_1 is the regular set driving $N \circ M$ into any of its accepting states.
4. Construct a labelled tree from the minimal derivatives, include the appropriate output on each branch of this tree.
5. Take the branches from the node corresponding to the last minimal derivative in the list of Step 3 and integrate the input derivatives of the terminal nodes of the branches. Also record the output expression corresponding to this integral; it can be read directly off the tree.
6. Combine the separate input integrals of Step 5 according to the integral manipulation rules 1 to 4 to form the complete integral of the last minimal derivative. Also record the corresponding output expression. Then delete this last minimal derivative.
7. Repeat Steps 5 and 6 for each minimal derivative in the reverse order of the list in Step 3 until all the integration is complete.

Denoting the output expression last recorded (which corresponds to state t_0) by R_2, we have $R_2 = M(R)$.

Justification:
First it should be pointed out that $M(R)$ *is* regular by Theorem 1. Now since Steps 3 through 7 are simply a restatement of the Smith and Yau algorithm in which one records not only the integration of the inputs but also that of the corresponding outputs, it follows from their paper that upon completion of the process R_1 will be a regular expression denoting the regular set driving $N \circ M$ into any of its accepting states. In precisely an analogous manner then, R_2 will be a regular expression denoting the acceptable output from $N \circ M$ when Σ_2^* is input, i.e.

$$R_2 = N \circ M(\Sigma_2^*) = M(N(\Sigma_2^*)) = M(R) .$$

The last equality following from Step 1.
Q.E.D.

Since a GSM is an a-GSM in which every state is an accepting state and a complete sequential machine (CSM, for short) is a GSM in which $g: Q \times \Sigma \to \Delta$, we can state the following result as a special case of the preceding algorithm.

Corollary 1:
There is an algorithm which, given an arbitrary non-empty regular expression R and either a GSM or a CSM, M, determines $M(R)$.

As an example, take $R = (a + b)(ca)^*$ and M to be the GSM of **Fig. 5**.

Step 1 has already been performed, resulting in the a-GSM of Fig. 4, which is called N in Algorithm 1. In Step 2 we construct $N \circ M$ as shown in **Fig. 6**.

Applying Steps 3 and 4 to this machine results in the labelled tree of minimal derivatives shown in **Fig. 7**.

Using this tree we then integrate the minimal derivatives in reverse order as specified in Steps 5, 6 and 7. (The corresponding output expression is recorded adjacent to the integration of the inputs).

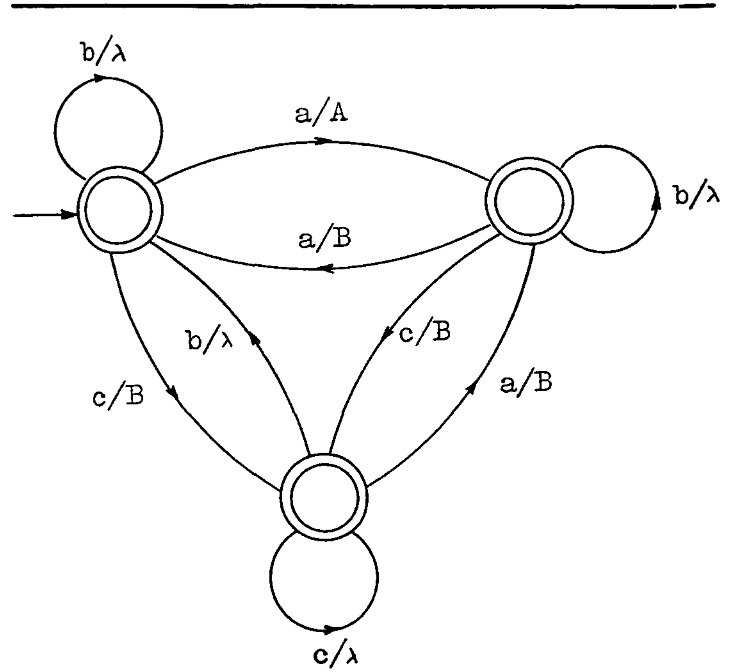


Fig. 5 GSM M of the example illustrating Algorithm 1

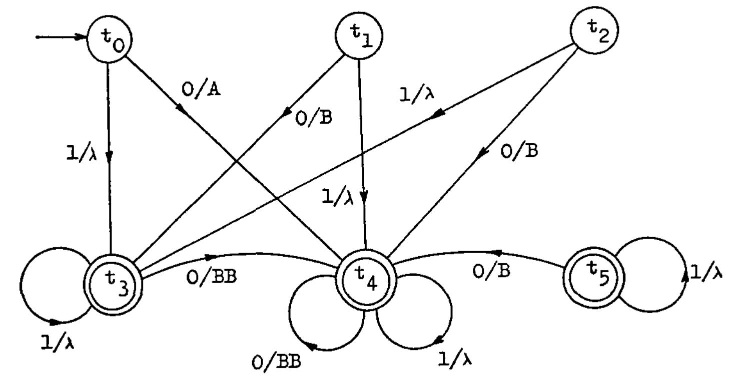


Fig. 6 Composite a-GSM $N \circ M$ obtained by Step 2 of Algorithm 1

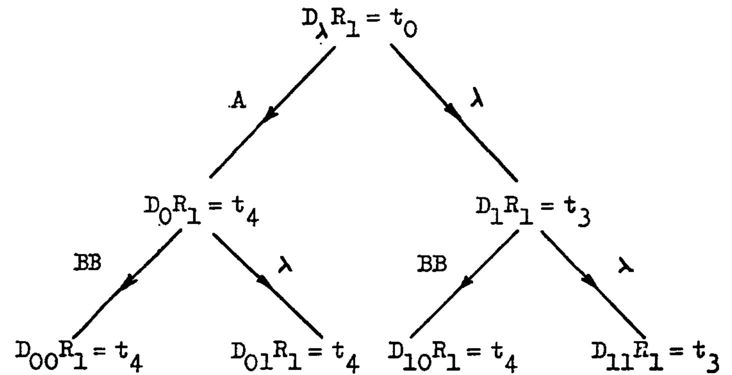


Fig. 7 The labelled tree of minimal derivatives for $N \circ M$ obtained by Steps 3 and 4 of Algorithm 1

$\int D_{10}R_1\, d0 = \int t_4\, d0 = 0t_4$	BBt_4
$\int D_{11}R_1\, d1 = \int t_3\, d1 = 1t_3$	λt_3
$t_3 = D_1R_1 = 0t_4 + 1t_3 + \lambda$	$BBt_4 + \lambda t_3 + \lambda$
$\int D_{00}R_1\, d0 = \int t_4\, d0 = 0t_4$	BBt_4
$\int D_{01}R_1\, d1 = \int t_4\, d1 = 1t_4$	λt_4

Downloaded from https://academic.oup.com/comjnl/article/17/4/349/443732 by guest on 19 April 2024

$t_4 = D_0R_1 = 0t_4 + 1t_4 + \lambda$	$BBt_4 + \lambda t_4 + \lambda$
$t_4 = (0 + 1)^*$	$(BB + \lambda)^*$
$t_3 = 0(0 + 1)^* + 1t_3 + \lambda$	$BB(BB + \lambda)^* + \lambda t_3 + \lambda$
$t_3 = 1^*(0(0 + 1)^* + \lambda)$	$\lambda^*(BB(BB + \lambda)^* + \lambda)$
$\int D_0R_1\, d0 = \int t_4\, d0 = 0t_4$	At_4
$\int D_1R_1\, d1 = \int t_3\, d1 = 1t_3$	λt_3
$t_0 = D_\lambda R_1 = 0t_4 + 1t_3$	$At_4 + \lambda t_3$
$t_0 = 0(0 + 1)^* + 1(1^*(0(0 + 1)^* + \lambda))$	$A(BB + \lambda)^* + \lambda\lambda^*(BB(BB + \lambda)^* + \lambda)$

Thus $M(R) = A(BB)^* + (BB)^*$ after removal of redundant 'λ''s.

It is clear that the algorithm is quite manageable by hand for many problems. For machine implementation one would need to modify the Smith-Yau program to record output expressions as well as short routines to produce the a-GSM generating R in Step 1 and to produce the composite a-GSM of Step 2.

Appendix

The notation, definitions and properties of regular sets and regular expressions are briefly reviewed here. It will be assumed that the reader is familiar with the usual definitions and properties of deterministic finite-state acceptors (FSA's, for short). Also some pertinent notions of the derivative and integral of a regular expression will be summarised.

Definition A1:
An *alphabet* is a finite non-empty set of symbols. The set of all finite sequences of symbols drawn from the alphabet Σ is denoted Σ^*. Any element of Σ^* is called a *word* (or *sequence*) over Σ. In particular, the word consisting of no symbols, the *null word*, denoted by λ is in Σ^*.

We now define some operations on sets of words.

Definition A2:
Let S_1 and S_2 be sets of words over the alphabet Σ, i.e. $S_1 \subseteq \Sigma^*$ and $S_2 \subseteq \Sigma^*$. Then in addition to the usual set-theoretic operations of union, intersection and complementation on S_1 and S_2 we define the *product* of S_1 and S_2, denoted by S_1S_2, to be

$$S_1S_2 = \{w \mid w = w_1w_2 \text{ such that } w_1 \in S_1 \text{ and } w_2 \in S_2\}$$

and the *star* of S_1, denoted by S_1^*, to be

$$S_1^* = \bigcup_{n=0}^{\infty} S_1^n ,$$

where we define inductively $S_1^0 = \{\lambda\}$, $S_1^1 = S_1$ and $S_1^n = S_1S_1^{n-1}$ for all $n \geqslant 2$.

One can then establish the following algebraic properties for sets of words: union and intersection are associative and commutative, product is associative but not commutative, product is distributive over union and intersection, λ serves as the multiplicative unity and $\emptyset$ (the empty set) serves as the multiplicative zero.

Definition A3:
A set of words, S, is called *regular* if there is an FSA which accepts exactly S.

One can conveniently characterise regular sets in the following way (due to Kleene): the class of regular sets is the least family which contains the finite sets and is closed under the operations of union, product and star. It is also true that intersection and complementation preserve regularity. This characterisation allows one to define the language of regular expressions which denote regular sets in a convenient manner.

Definition A4:
A *regular expression* over the alphabet $\Sigma = \{a_0, a_1, \ldots, a_{n-1}\}$ is defined inductively as follows:

1. The symbols $a_0, a_1, \ldots, a_{n-1}$ are regular expressions as are $\emptyset$ and λ.
2. If R_1 and R_2 are regular expressions, then so are $R_1 + R_2$, R_1R_2 and R_1^*.
3. Nothing else is a regular expression unless it follows from repeated applications of 1 and 2.

The connection between regular expressions and the regular sets they denote is made by merely associating '+' with '$\cup$' and interpreting the other operations as they were defined above. The terms 'regular set' and 'regular expression', while formally denoting different objects are many times used interchangeably in the future if no confusion will result.

In the description of the algorithm presented in this paper the notions of the derivative, the integral, and the complete integral of a regular expression are used. We now briefly describe these terms; see Smith and Yau (1972) for a more thorough treatment.

Definition A5:
Given a set of words S over the alphabet Σ and a word w over Σ, the *derivative of* S with respect to w, denoted by D_wS, is defined to be

$$D_wS = \{x \mid wx \in S\} .$$

Definition A6:
Given a set of words S over the alphabet Σ which is a derivative of the regular set R and a symbol $a_i \in \Sigma$, the *integral of* S with respect to a_i, denoted by $\int S\, da_i$, is defined to be

$$\int S\, da_i = \{S' \mid S' = a_iS, S' \subseteq R\} .$$

The integral of S with respect to a non-null word $w = a_{i_1}a_{i_2}, \ldots, a_{i_n}$, each $a_{i_j} \in \Sigma$ is defined recursively to be

$$\int \cdots \left[\int \left(\int S\, da_{i_n} \right) da_{i_{n-1}} \right] \cdots da_{i_1} = a_{i_1} \ldots a_{i_{n-1}}a_{i_n}S \subseteq R .$$

Finally, before we present the definition of the complete integral of a regular expression we must define the λ-integral.

Definition A7:
The *λ-integral*, denoted by Z, for the complete integral D_wR is defined to be

$$Z = \begin{cases} \lambda \text{ if } w \in R \\ \emptyset \text{ otherwise.} \end{cases}$$

Definition A8:
Given the regular expressions over the alphabet Σ $D_{a_i}R$, one for every $a_i \in \Sigma$, the *complete integral over* Σ is defined to be the regular expression

$$R = D_\lambda R = \sum_{i=0}^{n-1} \int D_{a_i}R\, da_i + Z .$$

If we substitute any word w over Σ for λ in the above equation, the complete integral D_wR over Σ becomes

$$D_wR = \sum_{i=0}^{n-1} \int D_{wa_i}R\, da_i + Z .$$

Downloaded from https://academic.oup.com/comjnl/article/17/4/349/443732 by guest on 19 April 2024

References

GINSBURG, S. (1966). *The Mathematical Theory of Context-Free Languages.* New York: McGraw-Hill Book Co.
GINSBURG, S., and HIBBARD, T. N. (1964). Solvability of machine mappings of regular sets to regular sets, *JACM*, Vol. 11, p. 302.
GINSBURG, S., and ROSE, G. F. (1963). Operations which preserve definability in languages, *JACM*, Vol. 10, p. 175.
GINSBURG, S., and SPANIER, E. H. (1966). Finite-turn pushdown automata, *J. Soc. for Industrial and Applied Math.*, Vol. 4, p. 429.
SMITH, L. W., and YAU, S. S. (1972). Generation of regular expressions for automata by the integral of regular expressions, *The Computer Journal*, Vol. 15, p. 222.
ULLIAN, J. S. (1967). Partial algorithm problems for context-free languages, *Information and Control*, Vol. 11, p. 80.

Book reviews

A Practical Approach to Computer Simulation in Business, by L. R. Carter and E. Huzan, 1973; 298 pages. (*George Allen and Unwin*, £5·95.)

The authors' purpose in writing this book is stated to be 'to introduce managers, systems analysts, industrial engineers and operational research workers to simulation in a practical manner'. To do this the book begins with chapters introducing the basic principles of simulation and its place in the field of management science and operation research. This is followed by a review of some necessary statistical concepts and a discussion of the methodology to be employed in building simulation models. Two chapters describing the use of FORTRAN and CSL (both for ICL 1900 series machines) for computer simulation are followed by three chapters which consider specific application of simulation models; in particular, queueing systems and forecasting and inventory control. The book concludes with a series of twenty-two appendices, the first fourteen of which are listings of various computer programs referred to in the text, and the remainder contain the usual statistical tables.

Unfortunately the content and coverage do not meet the stated objectives and the book falls into the trap of trying to be all things to all men. The first four chapters do provide a useful review of current management science techniques and the potential areas for the application of simulation methods. However, too often the authors indulge in rather sweeping statements for which no explanation is given, and although they laudably provide many examples as an integral part of the text, the authors' choice and discussion of the examples tends to be at a very low level (for example, on page 23 the algebraic operators for greater than or equal to, and less than or equal to, are explained). Despite this I feel that these chapters do form a useful introduction for managers, but I am afraid that for many systems analysts, industrial engineers and operational researchers the content and exposition will be far too low.

As in many books of this type, the inclusion of chapters on specific programming languages is a failure. The manager seeking an introduction to the subject does not need to acquire computer programming capabilities; the professional will either already possess such a capability or, if not, will find the material in these chapters inadequate for any real applications (even assuming he is using ICL equipment), and will have to have recourse to the usual programming texts. What the book lacks from the point of view of the professional is a rigorous treatment of simulation methodology (particularly emphasising the experimentation and inference aspects of simulation), together with a comparison of the usual high-level scientific programming languages with several of the more common simulation languages, from the particular viewpoint of implementing a simulation model (i.e. generating random variables; handling standard theoretical statistical distributions or user given discrete distributions; automatic recording of simulation variables and tabulation or plotting of them; automatic handling of simulation entities and their attributes; and so on).

To summarise, the book contains much that is useful and relevant, but fails as a text suitable for both managers and professionals. Managers will find the first four chapters useful, but there are other better (and cheaper) introductory texts on simulation. For the professional, he will require a text which has a much more rigorous and comprehensive approach to simulation methodology and its implementation.

J. R. EATON (London)

Integrated circuits in digital electronics, by A. Barna and D. I. Porat, 1973; 483 pages. (*John Wiley*, £11·25.)

Because all integrated circuits are encapsulated in standard package forms and their internal arrangements are not only highly complicated but quite inaccessible to the user, it is necessary to describe them in great detail to market them. For this reason the manufacturers of microcircuits have all gone to considerable expense and commendable effort to publish the most intimate details of their wares. Not only do they publish the circuits and performance specifications in great detail but most manufacturers back up these descriptions with Application Notes and most careful instructions for the use of the devices. These are usually given to users and potential users gratis. Some manufacturers have also produced truly useful books on the use and applications of their products; some of these are also given away and those that must be paid for are good value for money.

This very expensive book—in comparison with those produced by manufacturers—contains little, if any, real information about microcircuits that is not available in the manufacturers' books. As a reference document it is not as good as they are.

As a teaching document likewise it is not very good. It has chapters on Number Systems, Combinational and Sequential Logic, Arithmetic Circuits, D to A and A to D Converters, and the like, which are normal contents of most text books on 'Logic'. These are no better than the corresponding chapters in these text books and add nothing of value to the presently available range. There are exercises and examples with answers to some of them, but so there are in most books on the subject.

Possibly it might be a good idea to combine the basic technique with its application in microcircuits in one book; that is obviously what is intended by the authors. It makes a thick and unnecessarily expensive book but it adds rather little to what is readily available elsewhere. For the sake of the record, technical libraries should have a copy for reference. I cannot in my heart recommend it to anyone else.

B. S. WALKER (Reading)

Downloaded from https://academic.oup.com/comjnl/article/17/4/349/443732 by guest on 19 April 2024