8

Recognition of Realizability of a Given Specification. Abstract Synthesis of Finite Automata and Sequential Machines

8.1. STATEMENT OF THE PROBLEM

The design of any specific automatic device involves several independent stages. Thus the designer starts by analyzing and then "idealizing" the operations required of the device. Here, the designer may obtain an idealization which specifies the problem in terms of discrete time and a finite number of variables, each assuming only a finite number of values. If that is the case, he may be able to employ a finite automaton or a sequential machine. We say "may be able" because not all problems, even if formulated in terms of discrete time and a finite number of variables, can be performed in a finite automaton or an s-machine. For example, these machines cannot "forecast" the state of the input, that is, they are unable to generate, at $t = p$, an output corresponding to an input at $t = p + 1$. But even if our specification calls for an output depending only on the preceding input states, there may not exist a machine embodying the specifications. We have seen this in our attempt to synthesize an automaton for representation of irregular events (Chapter 7).

Thus the second design stage involves finding out whether finite automata or s-machines are suitable for a given task, a problem we shall denote as that of *recognition of realizability* of a given specification (or simply the *recognition problem*).

Assuming that our specification is realizable in either of these discrete devices, we enter stage three, where we determine their basic tables. This is the stage of *abstract synthesis*.

After abstract synthesis, the most important phase of the design is ended. Before it, the designer deals only with the specification of the ultimate automatic device, that is, with given input and output sequences and the specified relationships between them. After the abstract synthesis stage, he has a table of the automaton (and of the converter, if an s-machine is involved), and in all subsequent stages works only with these tables. He now simplifies them as much as possible, selects the best overall means of their realization, and solves the practical problems related to specific technology of the selected devices. This brings the logic design to an end.

With the exception of Chapter 7, we have always assumed that the automaton and converter tables were given, and did not bother with the problem of how these tables were obtained. Now we shall deal with the techniques for generating these tables starting from specifications for the device; that is, we shall deal with problems of recognition and the abstract synthesis.

In speaking of "specification of the device," we assumed that the reader has an intuitive understanding of what is involved. Now, however, we must define just what this sentence means.

In all cases "specification of the device" means the definition of the correspondences between the given input and output sequences. The simplest case is that of finite number of given input and output sequences, where "specification of the device" assumes the very definite meaning of enumeration of all the given sequences and all the correspondences between them. This type of specification is treated in Section 8.2.

The situation is much more complicated in the general case, when the number of given sequences (and, consequently, their lengths) can be infinite. Here it is impossible to employ enumeration, and the infinite input and output sequences, as well as the correspondences between them are specified by means of a *defining language*.

The problems of recognition and abstract synthesis may be formulated as follows: we have a defining language and we have described the sets of input and output sequences and the correspondences between them in this language. Now we must find an algorithm (that is, a procedure) for determining whether there exists an automaton (or s-machine) capable of setting up such correspondences, and whether there exists an algorithm capable of generating the basic table of the automaton (and the converter), if such machines do exist.

It turns out, however, that the very ability to find such algorithms depends on the defining language. If this language is too broad, then there are no such algorithms; that is, the problems of recognition and abstract synthesis are algorithmically unsolvable (see Section 8.3). Thus there is the problem of narrowing the language in which the design specification is stated. One of such narrow languages—the language of regular formulas—is described in Section 8.4, where it is shown how, starting from a given regular formula, one can synthesize a relatively economical (insofar as the number of states is concerned) s-machine which realizes the specification. The problem as to whether such a machine is at all possible does not arise here, since the language can only describe events that are realizable.

8.2. THE CASE WHERE THE SPECIFICATION ENUMERATES THE REQUIRED INPUT-OUTPUT CORRESPONDENCES

Assume we want to synthesize an s-machine specified as follows: We are given a finite number of tapes (which can be of different, but finite, lengths), with a blank "$\varkappa$" row, for example, the four tapes of Tables 8.1 - 8.4. The required s-machine must realize these tapes, starting from a given initial state $\varkappa^0$.

This specification does not say anything about other possible tapes (at other input sequences) of this same s-machine. If there are no specific instructions to this effect, we shall assume that no other input sequences are possible or, which amounts to the same, that all other tapes may be arbitrarily chosen. The specification may also include additional conditions, for example, the requirement that any other tape must contain some specific symbol (for example, λ_0) at all sampling instants, starting from the instant in which the ρ row of such

Table 8.1

Discrete moment	0	1	2	3	4	5
ρ	ρ_1	ρ_2	ρ_0	ρ_1	ρ_3	—
$\varkappa$						
λ	λ_1	λ_2	λ_1	λ_1	λ_3	λ_1

Table 8.2

Discrete moment	0	1
ρ	ρ_0	ρ_3
$\varkappa$		
λ	λ_2	λ_2

Table 8.3

Discrete moment	0	1	2
ρ	ρ_3	ρ_1	—
$\varkappa$			
λ	λ_3	λ_1	λ_2

Table 8.4

Discrete moment	0	1	2
ρ	ρ_2	ρ_0	—
$\varkappa$			
λ	λ_2	λ_1	λ_1

sampling instants, starting from the instant in which the ρ row of such a tape begins to differ from the ρ rows of the tapes enumerated in the specification. If the main tapes are those of Tables 8.1 - 8.4, then this additional condition could, for example, lead to an s-machine having tapes shown in Tables 8.5 - 8.7 (where the instants of generation of λ_0 are marked off with heavy lines).

Table 8.5

Discrete moment	0	1	2	3	4	5
ρ	ρ_1	ρ_2	ρ_0	ρ_3	ρ_1	—
$\varkappa$						
λ	λ_1	λ_2	λ_1	λ_1	λ_0	λ_0

Table 8.6

Discrete moment	0	1	2
ρ	ρ_3	ρ_2	—
$\varkappa$			
λ	λ_3	λ_1	λ_0

Table 8.7

Discrete moment	0	1	2	3	4	5
ρ	ρ_0	ρ_3	ρ_1	ρ_3	ρ_1	—
$\varkappa$						
λ	λ_2	λ_2	λ_3	λ_1	λ_0	λ_0

Because we stipulated that all the given tapes must start with $\varkappa^0$, our specification may prove to be an impossible one. For example, consider Tables 8.8 and 8.9. Obviously, there is no s-machine which has both these tapes: for this machine would generate, from the same initial state $\varkappa^0$ and the same input sequence $\rho_1\rho_2\rho_0$, two different outputs (λ_1 in the case of 8.8, and λ_3 in the case of 8.9). Such an operation cannot be expected of an s-machine, which by definition is a determinate device. The specification is thus *contradictory.*

Now it is clear that before proceeding with the synthesis, the designer must check the specification for contradictions. He does this by inspecting sections of the given tapes, as shown below. The specification is not contradictory if, and only if, no two sections show differing outputs at identical inputs.

Table 8.8

Discrete moment	0	1	2	3	4	5
ρ	ρ_1	ρ_2	ρ_0	ρ_3	ρ_1	—
$\varkappa$	$\varkappa^0$					
λ	λ_1	λ_2	λ_1	λ_1	λ_3	λ_1

Table 8.9

Discrete moment	0	1	2	3	4	5	6
ρ	ρ_1	ρ_2	ρ_0	ρ_3	ρ_2	ρ_1	—
$\varkappa$	$\varkappa^0$						
λ	λ_1	λ_2	λ_1	λ_3	λ_1	λ_2	λ_1

Thus, in the example of tapes 8.8 and 8.9, the tapes are split up as follows:

for $p=1$

Discrete moment	0	1
ρ	ρ_1	—
$\varkappa$	$\varkappa^0$	
λ	λ_1	λ_2

Discrete moment	0	1
ρ	ρ_1	—
$\varkappa$	$\varkappa^0$	
λ	λ_1	λ_2

for $p=2$

Discrete moment	0	1	2
ρ	ρ_1	ρ_2	—
$\varkappa$	$\varkappa^0$		
λ	λ_1	λ_2	λ_1

Discrete moment	0	1	2
ρ	ρ_1	ρ_2	—
$\varkappa$	$\varkappa^0$		
λ	λ_1	λ_2	λ_1

for $p=3$

Discrete moment	0	1	2	3
ρ	ρ_1	ρ_2	ρ_0	—
$\varkappa$	$\varkappa^0$			
λ	λ_1	λ_2	λ_1	λ_1

Discrete moment	0	1	2	3
ρ	ρ_1	ρ_2	ρ_0	—
$\varkappa$	$\varkappa^0$			
λ	λ_1	λ_2	λ_1	λ_3

for $p=4$

Discrete moment	0	1	2	3	4
ρ	ρ_1	ρ_2	ρ_0	ρ_3	—
$\varkappa$	$\varkappa^0$				
λ	λ_1	λ_2	λ_1	λ_1	λ_3

Discrete moment	0	1	2	3	4
ρ	ρ_1	ρ_2	ρ_0	ρ_3	—
$\varkappa$	$\varkappa^0$				
λ	λ_1	λ_2	λ_1	λ_3	λ_1

for $p=5$

Discrete moment	0	1	2	3	4	5
ρ	ρ_1	ρ_2	ρ_0	ρ_3	ρ_1	—
$\varkappa$	$\varkappa^0$					
λ	λ_1	λ_2	λ_1	λ_1	λ_3	λ_1

Discrete moment	0	1	2	3	4	5
ρ	ρ_1	ρ_2	ρ_0	ρ_3	ρ_2	—
$\varkappa$	$\varkappa^0$					
λ	λ_1	λ_2	λ_1	λ_3	λ_1	λ_2

The contradiction becomes apparent after $p=3$.

Assume that we start with a noncontradictory specification so that we can immediately proceed with the synthesis, that is, develop the tables of the finite automaton and of the converter of the s-machine. If we are given all the $\varkappa$ rows on the given tapes, than we can pick out the triads directly from the tapes and write out at least some of the squares of these tables. The other squares could be filled out arbitrarily, if no additional conditions are imposed on the machine, or in some other way, if there are such conditions. Thus the synthesis of an s-machine reduces to filling out the $\varkappa$ rows of the tapes in a noncontradictory manner.

If the specification can be realized in some s-machine A, then it can also be realized in any s-machine B which can substitute for A. If all we require is some machine realizing the specification, that is, if we impose no additional requirements on the s-machine, then the solution is trivial, and may involve one of the following two methods:

a) The $\varkappa$ rows may be filled with nonrecurring $\varkappa_i$, the number of the different states $\varkappa_i$ thus obtained being equal to the sum of lengths of the (given) tapes.

b) We can construct an automaton with an output converter which represents the specific events "recorded" on the (given) tapes. Thus, we extract from the tapes all those input sets G_1, G_2, and so on, which generate outputs λ_1, λ_2, and so on, respectively. Then, using methods of Chapter 7, we synthesize automata I, II, and so on, respectively representing the specific events G_1, G_2, and so on by outputs of 1. We now connect the outputs of these automata to a converter generating an output of λ_1 if there is a 1 on the first converter input, λ_2 with a 1 on the second input, and so on.

The s-machines synthesized by either of these methods usually have an extremely large number of states. Therefore, quite often one is faced with a much more difficult problem, where one wants to synthesize an s-machine conforming to a given specification but having fewer states than any other machine also conforming to this specification.

The solution is usually divided into two phases: 1) synthesis of some (any) machine conforming to specifications; and 2) its minimization, that is, the derivation, from this preliminary version, of another s-machine which also conforms to the specification but which has the least possible number of states k.

The *minimization problem* of phase 2 is discussed in Chapter 9. However, the methods of that chapter are difficult to apply if the s-machine of phase 1 has many states, a condition produced by the use of the above trivial methods in phase 1. For this reason one resorts, if possible, to other phase 1 techniques: these give s-machines which, even though not minimal, are *a priori* known to have a smaller number of states. We shall describe one such method.

Let us begin with the following example. Suppose we want to synthesize an s-machine which, starting from the same state $\varkappa^0 = \varkappa_0$, will realize the two tapes of Tables 8.10 and 8.11. The tapes generated at other input sequences can be arbitrary.

Let us prepare a blank form for the tables of the automaton and the converter (Table 8.12 for the automaton and Table 8.13 for the converter). Since the ultimate number of the states is still unknown, the number of the rows in these tables is still undetermined. For the time being, we have only one line—for $\varkappa_0$.

Table 8.10

Discrete moment	0	1	2	3	4	5	6
ρ	ρ_0	ρ_3	ρ_2	ρ_3	ρ_0	ρ_2	ρ_3
$\varkappa$	$\varkappa_0$						
λ	λ_1	λ_1	λ_2	λ_2	λ_1	λ_2	λ_2

Table 8.11

Discrete moment	0	1	2	3	4	5	6
ρ	ρ_3	ρ_2	ρ_0	ρ_0	ρ_1	ρ_3	ρ_0
$\varkappa$	$\varkappa_0$						
λ	λ_1	λ_2	λ_1	λ_1	λ_2	λ_2	λ_1

In further filling out the $\varkappa$ row, we must always check whether the new entry does not contradict the preceding one; that is, that it does

Table 8.12
Automaton

$\varkappa$ \ ρ	ρ_0	ρ_1	ρ_2	ρ_3
$\varkappa_0$				

Table 8.13
Converter

$\varkappa$ \ ρ	ρ_0	ρ_1	ρ_2	ρ_3
$\varkappa_0$				

not produce contradictory triads in the automaton or impose incompatible requirements on the converter.

In the first square of row $\varkappa$ of Table 8.10 we write $\varkappa_0$ (in accordance with the specification). There is also no reason why $\varkappa_0$ cannot be entered in the second square of this tape. We also write $\varkappa_0$ in the corresponding square of the basic table of the automaton (Table 8.12). We fill the corresponding squares ($\varkappa_0$, ρ_0) and ($\varkappa_0$, ρ_3) of the converter table with the symbol λ_1 (from columns 0 and 1 of Table 8.10). Again we observe that nothing prevents us from entering the same symbol $\varkappa_0$ into the column 2 of Table 8.10 and into the corresponding squares of Tables 8.12 and 8.13. This results in Tables 8.14 - 8.16.

Table 8.14
Tape

Discrete moment	0	1	2	3	4	5	6
ρ	ρ_0	ρ_3	ρ_2	ρ_3	ρ_0	ρ_2	ρ_3
$\varkappa$	$\varkappa_0$	$\varkappa_0$	$\varkappa_0$				
λ	λ_1	λ_1	λ_2	λ_2	λ_1	λ_2	λ_2

Table 8.15
Basic Table of the Automaton

$\varkappa$ \ ρ	ρ_0	ρ_1	ρ_2	ρ_3
$\varkappa_0$	$\varkappa_0$			$\varkappa_0$

Table 8.16
Converter Table

$\varkappa$ \ ρ	ρ_0	ρ_1	ρ_2	ρ_3
$\varkappa_0$	λ_1		λ_2	λ_1

So far, we entered $\varkappa_0$ without producing contradictions. Let us try to enter it in the next (fourth) square of the tape (Table 8.14). This produces no contradictions in the automaton table since this symbol goes into a blank square, but an attempt to write in λ_2 into the square $(\rho_3, \varkappa_0)$ of the converter does lead to contradiction since this square already contains λ_1. We have no other alternative but to enter a new symbol $\varkappa_1$ into the fourth square of the tape (Table 8.14). This, of course, gives new rows in the tables of the automaton and the converter, where we enter the corresponding symbols.

Now we try to enter $\varkappa_0$ into the next (fifth) square of the tape: there is no contradiction. Had there been one, then we would have had to try $\varkappa_1$ and, if this led to a contradiction, we would have to introduce $\varkappa_2$ and start a new row in the tables. We repeat this procedure all along the tape of Table 8.14 and we thus complete its $\varkappa$ row. Now we turn to the second tape (Table 8.11) and fill its $\varkappa$ row in the same manner. Here we must make sure that the symbol being entered not only does not contradict the previous entries on this tape, but that it does not contradict those of the first tape. We then have complete tables for the automaton and the converter (Tables 8.17 - 8.20). Now these tables have blank squares which may be filled in any desired manner, since we have stiuplated at the outset that the tapes obtained at input sequence other than the given ones are arbitrary.

Table 8.17

First Tape

Discrete moment	0	1	2	3	4	5	6
ρ	ρ_0	ρ_1	ρ_2	ρ_3	ρ_0	ρ_2	ρ_3
$\varkappa$	$\varkappa_0$	$\varkappa_0$	$\varkappa_0$	$\varkappa_1$	$\varkappa_0$	$\varkappa_0$	$\varkappa_1$
λ	λ_1	λ_1	λ_2	λ_2	λ_1	λ_2	λ_2

Table 8.18

Second Tape

Discrete moment	0	1	2	3	4	5	6
ρ	ρ_3	ρ_2	ρ_0	ρ_0	ρ_1	ρ_3	ρ_0
$\varkappa$	$\varkappa_0$	$\varkappa_0$	$\varkappa_1$	$\varkappa_0$	$\varkappa_0$	$\varkappa_1$	$\varkappa_1$
λ	λ_1	λ_2	λ_1	λ_1	λ_2	λ_2	λ_1

Table 8.19

Basic Table of the Automaton

$\varkappa$ \ ρ	ρ_0	ρ_1	ρ_2	ρ_3
$\varkappa_0$	$\varkappa_0$	$\varkappa_1$	$\varkappa_1$	$\varkappa_0$
$\varkappa_1$	$\varkappa_0$	—	—	$\varkappa_1$

Table 8.20

Converter Table

$\varkappa$ \ ρ	ρ_0	ρ_1	ρ_2	ρ_3
$\varkappa_0$	λ_1	λ_2	λ_2	λ_1
$\varkappa_1$	—	—	—	λ_2

The above example has been chosen so as to present no difficulties in completing the $\varkappa$ rows. However, consider the tapes of Tables 8.21 and 8.22. We leave the intermediate details to the reader, and shall discuss only the final results. Thus, in tape 1, we can enter $\varkappa_0$ into the second and the third squares without raising any contradictions,

Table 8.21

First Tape

Discrete moment	0	1	2	3	4	5	6
ρ	ρ_0	ρ_1	ρ_2	ρ_1	ρ_0	ρ_2	ρ_1
$\varkappa$	$\varkappa_0$						
λ	λ_2	λ_1	λ_2	λ_2	λ_2	λ_1	λ_0

Table 8.22

Second Tape

Discrete moment	0	1	2	3	4
ρ	ρ_2	ρ_1	ρ_2	ρ_0	ρ_2
$\varkappa$	$\varkappa_0$				
λ	λ_2	λ_0	λ_2	λ_2	λ_2

but since $\varkappa_0$ in the fourth square produces a contradiction in the converter table, we must write $\varkappa_1$ in this square. A $\varkappa_0$ in the fifth square does not, in itself, lead to a contradiction, but we then must have $\varkappa_0$ in the sixth square to avoid a contradiction in the automaton table [since the ($\varkappa_0$, ρ_0) combination was already used in the first column of the tape and required a $\varkappa_0$ in the following column]. But $\varkappa_0$ in the sixth square contradicts the converter table. We must, therefore, go back to the fifth square and try $\varkappa_1$. This gives no contradiction, and we can thus try $\varkappa_1$ in the sixth square. No (ρ_2, $\varkappa_1$) combination has yet been encountered and, therefore, from the point of view of the table of the automaton, we can use any symbol in the seventh tape square. However, $\varkappa_0$ and $\varkappa_1$ cannot be used because combinations $(\rho_1, \varkappa_0)$ and $(\rho_1, \varkappa_1)$ already specify entries other than λ_1 in the converter table. Therefore, we use a new symbol $\varkappa_2$ in the seventh square.

Now let us examine tape 2 (Table 8.22). The (ρ_2, $\varkappa_0$) combination has already been encountered in the third column of tape 1; therefore, to avoid contradictions in the automaton table, we can only use $\varkappa_1$ in the second square of tape 2. But this causes a contradiction in the converter table, so we have no alternative but to go back to the third square of the tape 1, remove the $\varkappa_0$, and fill in $\varkappa_3$. Now we can proceed with the completion of the tape 2 without altering tape 1, resolving only contradictions that may arise in the same manner as when completing tape 1. No states other than $\varkappa_0$, $\varkappa_1$, $\varkappa_2$ and $\varkappa_3$ need to be used in this example, which finally results in Tables 8.23 - 8.26.

Table 8.23

First Tape

Discrete moment	0	1	2	3	4	5	6
ρ	ρ_0	ρ_1	ρ_2	ρ_1	ρ_0	ρ_2	ρ_1
$\varkappa$	$\varkappa_0$	$\varkappa_0$	$\cancel{\varkappa_0}$ $\varkappa_3$	$\cancel{\varkappa_0}$ $\varkappa_1$	$\cancel{\varkappa_0}$ $\varkappa_1$	$\cancel{\varkappa_0}$ $\varkappa_1$	$\cancel{\varkappa_0}$ $\cancel{\varkappa_1}$ $\varkappa_2$
λ	—	λ_1	λ_2	λ_2	λ_2	λ_1	λ_0

Table 8.24

Second Tape

Discrete moment	0	1	2	3	4
ρ	ρ_2	ρ_1	ρ_2	ρ_0	ρ_2
$\varkappa$	$\varkappa_0$	$\cancel{\varkappa_1}$ $\varkappa_2$	$\cancel{\varkappa_3}$ $\varkappa_0$	$\cancel{\varkappa_1}$ $\varkappa_2$	$\cancel{\varkappa_1}$ $\varkappa_2$
λ	—	λ_0	λ_2	λ_2	λ_2

Table 8.25

Basic Table of the Automaton

$\varkappa$ \ ρ	ρ_0	ρ_1	ρ_2
$\varkappa_0$	$\varkappa_0$	$\varkappa_3$	$\varkappa_2$
$\varkappa_1$	$\varkappa_1$	$\varkappa_1$	$\varkappa_2$
$\varkappa_2$	$\varkappa_2$	$\varkappa_0$	—
$\varkappa_3$	—	—	$\varkappa_1$

Table 8.26

Converter Table

$\varkappa$ \ ρ	ρ_0	ρ_1	ρ_2
$\varkappa_0$	λ_1	λ_1	λ_2
$\varkappa_1$	λ_2	λ_2	λ_1
$\varkappa_2$	λ_2	λ_0	λ_2
$\varkappa_3$	—	—	λ_2

Rather than replacing the $\varkappa_0$ in the third square of tape 1 with a new symbol $\varkappa_3$, we could have tried to use one of the old ones ($\varkappa_1$ or $\varkappa_2$), altering, if necessary, the other squares of tape 1. This would have given other tables for the automaton and converter, and these might have had a different number of rows (that is, states).

The method demonstrated above is not a regular, straightforward procedure, and shows the complexities which may arise in phase 1 of the synthesis, where we are merely trying to obtain *any* s-machine satisfying the stipulated conditions.

Let us now develop a regular method for uniform synthesis of relatively economical s-machines realizing any given noncontradictory finite set of finite tapes. We shall confine ourselves to the case where the tapes generated at input sequences other than those stipulated in the specification may be arbitrary. Again we shall start with the blanks for the tables of the automaton and the converter, and we shall increase the number of rows in these tables as new states κ are introduced. Assume, however, that the given tapes and the two tables are already partly filled. We shall say that these already-present entries are correct if they meet the following conditions:

1. The entries in the tapes and the tables do not conflict; that is, the tapes contain no triads producing contradictions in the automaton and the converter and, conversely, the only filled positions in the automaton and converter tables are those which correspond to triads already present on the tapes.

2. The last column of each tape contains a (ρ, κ) pair which defines a still empty square in the automaton table.

3. If two or more of these last (ρ, κ) pairs contain identical symbols ρ, and if the corresponding tapes show identical ρ's during one or more subsequent sampling instants, then these tapes must also show identical symbols λ during these instants.

For example, Tables 8.27 - 8.30 are correctly filled.

Table 8.27

First Tape

Discrete moment	0	1	2	3	4	5	6
ρ	ρ_2	ρ_1	ρ_3	ρ_3	ρ_2	ρ_1	ρ_3
κ	κ_0	κ_0	κ_1	κ_0			
λ	λ_2	λ_1	λ_3	λ_1	λ_3	λ_1	λ_1

Table 8.28

Second Tape

Discrete moment	0	1	2	3	4	5	6	7
ρ	ρ_1	ρ_2	ρ_1	ρ_3	ρ_1	ρ_2	ρ_2	ρ_2
κ	κ_0	κ_1	κ_1	κ_0				
λ	λ_1	λ_2	λ_3	λ_1	λ_2	λ_3	λ_1	λ_3

Table 8.29

Basic Table of the Automaton

$\kappa \backslash \rho$	ρ_1	ρ_2	ρ_3
κ_0	κ_1	κ_0	
κ_1	κ_0	κ_1	κ_0

Table 8.30

Converter Table

$\kappa \backslash \rho$	ρ_1	ρ_2	ρ_3
κ_0	λ_1	λ_2	λ_1
κ_1	λ_3	λ_2	λ_3

However, the tables of the following three examples are incorrectly filled.

Example 1 (Tables 8.31 - 8.34) violates condition 1:

Table 8.31

First Tape

Discrete moment	0	1	2	3	4	5	6	7
ρ	ρ_1	ρ_3	ρ_2	ρ_1	ρ_3	ρ_2	ρ_4	ρ_1
$\varkappa$	$\varkappa_0$	$\varkappa_0$	$\varkappa_1$	$\varkappa_0$	$\varkappa_1$			
λ	λ_1	λ_3	λ_4	λ_1	λ_4	λ_3	λ_1	λ_1

Table 8.32

Second Tape

Discrete moment	0	1	2	3	4	5	6
ρ	ρ_4	ρ_2	ρ_3	ρ_4	ρ_3	ρ_2	ρ_4
$\varkappa$	$\varkappa_0$	$\varkappa_1$	$\varkappa_0$	$\varkappa_1$			
λ	λ_2	λ_3	λ_3	λ_1	λ_2	λ_3	λ_2

Table 8.33

Basic Table of the Automaton

$\varkappa$ \ ρ	ρ_1	ρ_2	ρ_3	ρ_4
$\varkappa_0$	$\cancel{\varkappa_0}$?		$\varkappa_1$	
$\varkappa_1$		$\varkappa_0$		

Table 8.34

Converter Table

$\varkappa$ \ ρ	ρ_1	ρ_2	ρ_3	ρ_4
$\varkappa_0$	λ_1		λ_3	λ_1
$\varkappa_1$		$\cancel{\lambda_4}$?	λ_4	

Example 2 (Tables 8.35 - 8.38) violates condition 2:

Table 8.35

First Tape

Discrete moment	0	1	2	3	4	5	6	7
ρ	ρ_2	ρ_1	ρ_3	ρ_2	ρ_3	ρ_1	ρ_1	ρ_1
$\varkappa$	$\varkappa_0$	$\varkappa_0$	$\varkappa_1$	$\varkappa_0$				
λ	λ_2	λ_1	λ_3	λ_2	λ_2	λ_2	λ_1	λ_2

Table 8.36

Second Tape

Discrete moment	0	1	2	3	4	5
ρ	ρ_3	ρ_3	ρ_2	ρ_1	ρ_3	ρ_1
$\varkappa$	$\varkappa_0$	$\varkappa_1$	$\varkappa_0$	$\varkappa_0$		
λ	λ_1	λ_3	λ_2	λ_1	λ_3	λ_1

Example 3 (Tables 8.39 - 8.41) violates condition 3:

Provided the specification is not contradictory, the initial (given) tapes which contain only $\varkappa_0$ for $t = 0$, together with a completely blank automaton table and a converter table with entries only in the squares corresponding to $t = 0$ for all the tapes, are one case of correct filling.

Table 8.37

Basic Table of the Automaton

$\varkappa \backslash \rho$	ρ_1	ρ_2	ρ_3
$\varkappa_0$	$\varkappa_1$	$\varkappa_0$	$\varkappa_1$
$\varkappa_1$			$\varkappa_0$

Table 8.38

Converter Table

$\varkappa \backslash \rho$	ρ_1	ρ_2	ρ_3
$\varkappa_0$	λ_1	λ_2	λ_1
$\varkappa_1$			λ_3

Table 8.39

First Tape

Discrete moment	0	1	2	3	4	5
ρ	ρ_2	ρ_1	ρ_3	ρ_3	ρ_2	ρ_1
$\varkappa$	$\varkappa_0$	$\varkappa_0$	$\varkappa_1$	$\varkappa_0$		
λ	λ_2	λ_1	λ_3	λ_1	λ_3	λ_2

Table 8.40

Second Tape

Discrete moment	0	1	2	3	4	5
ρ	ρ_1	ρ_2	ρ_1	ρ_3	ρ_2	ρ_1
$\varkappa$	$\varkappa_0$	$\varkappa_1$	$\varkappa_1$	$\varkappa_0$		
λ	λ_1	λ_2	λ_3	λ_1	λ_3	λ_2

Table 8.41

Third Tape

Discrete moment	0	1	2	3	4
ρ	ρ_1	ρ_1	ρ_3	ρ_2	ρ_1
$\varkappa$	$\varkappa_0$	$\varkappa_1$	$\varkappa_0$		
λ	λ_1	λ_3	λ_1	λ_2	λ_3

Here conditions 1 and 2 are automatically met, and the noncontradictory nature of the specification is precisely what condition 3 is about.

We shall now describe the algorithm for entering states $\varkappa$ in the blanks of the tape, assuming that the tape already contains some (not necessarily initial) correct entries of $\varkappa_0, \varkappa_1, \ldots, \varkappa_s$. We start by numbering the (given) tapes in any desired order. Then:

1. We turn to the first *blank* square of row $\varkappa$ in tape 1 and try to enter $\varkappa_0$, checking whether this does not raise contradictions in the converter table. If there is a contradiction, we try $\varkappa_1$, again checking for contradiction. If none of the symbols $\varkappa_0, \varkappa_1, \ldots, \varkappa_s$ removes the contradiction, we introduce a new symbol $\varkappa_{s+1}$ and add a new row to the tables of the automaton and the converter. Let $\varkappa'$ be the first symbol which produces no contradictions in the converter. If $\varkappa' = \varkappa_{s+1}$, then we make corresponding entries in the tape and the tables and proceed immediately to step 3 of the algorithm. If, however, $\varkappa' = \varkappa_k$ $(0 \leqslant k \leqslant s)$, we again make the corresponding entries and proceed to step 2.

2. Turning to the automaton table (now supplemented with a new square in accordance with step 1), we ascertain whether we can continue filling tape 1 (without filling in new squares in the automaton table). If this is possible, we keep on filling the tape, making sure that no contradictions arise in the converter table. If no contradictions occur, we keep on filling the tape until we encounter a blank square in the automaton table or until tape 1 is completed, whereupon we proceed to step 3 of the algorithm; If a contradiction with the converter table does occur, we return to that square of tape 1 where at the end of step 1 we wrote $\varkappa' = \varkappa_k$: we erase $\varkappa_k$ from all of our tables, and we also erase the other entries associated with it and made in step 1. We then continue the search for a suitable $\varkappa$ as per step 1, starting this search with $\varkappa_{k+1}$. After a finite number of trials, we must be able to proceed with step 3 of the algorithm (because if $\varkappa'$ is not in the sequence $\varkappa_0, \varkappa_1, \ldots, \varkappa_s$, then we must introduce a new symbol $\varkappa_{s+1}$).

3. Assume that the procedures of steps 1 and 2 finally give a suitable, noncontradictory symbol $\varkappa'' = \varkappa_m$ (where $k \leqslant m \leqslant s+1$). We now return to the entries already present on tape 1 at the start of our procedure, and we take the $(\rho, \varkappa)$ pair in the last correctly filled column. We then check the last correctly filled columns of the remaining tapes for the presence of this pair. If no such pairs are present, we proceed to step 4 of the algorithm. If, however, there are such pairs, then we try to continue filling, as per step 2, each tape in which the last "correct" $(\rho, \varkappa)$ pair coincides with the last "correct" $(\rho, \varkappa)$ pair of tape 1.* Here, there are two possibilities: a) we may be able to fill these tapes to the end (that is, until we reach a blank square in the automaton table, or until the tape is completed), whereupon we proceed to step 4 of the algorithm, or b) we may arrive at a contradiction with the converter table. If the latter is the case, we return to step 1 of the algorithm, drop symbol $\varkappa_m$, erase all the entries associated with it, and continue the search for a suitable $\varkappa'$ as per step 1, starting with $\varkappa_{m+1}$. After a finite number of trials, we must be able to proceed to step 4 of the algorithm because, if no other $\varkappa'$ is found, we will use $\varkappa_{s+1}$ which definitely allows us to go to step 4.

4. We check the tapes for conformity with condition 3 for correct entries. If this condition is met, we have again arrived at correct entries. We can then return to algorithm step 1, and continue filling the tapes and tables. However, if the tape does not meet condition 3, we erase all tape and table entries associated with $\varkappa'' = \varkappa_m$, and

*If a new symbol $\varkappa_{s+1}$ had been introduced, then the maximum possible advance is one square.

continue the search for a suitable $\varkappa'$ as per step 1, starting the search with $\varkappa_{m+1}$. However, if $\varkappa'' = \varkappa_{s+1}$, then the check of step 4 will always show that condition 3 holds; thus, this check can be omitted.

The reader is advised to use this algorithm to synthesize the automaton realizing the tapes of Tables 8.42 - 8.44. In this case the algorithm must be used six times and finally gives the tapes and Tables 8.45 - 8.49.

Table 8.42

First Tape

Discrete moment	0	1	2	3	4	5
ρ	ρ_2	ρ_1	ρ_1	ρ_3	ρ_2	ρ_1
$\varkappa$	$\varkappa_0$					
λ	λ_1	λ_3	λ_2	λ_1	λ_2	λ_3

Table 8.43

Second Tape

Discrete moment	0	1	2
ρ	ρ_1	ρ_3	ρ_2
$\varkappa$	$\varkappa_0$		
λ	λ_2	λ_1	λ_2

Table 8.44

Third Tape

Discrete moment	0	1	2	3
ρ	ρ_2	ρ_3	ρ_1	ρ_2
$\varkappa$	$\varkappa_0$			
λ	λ_1	λ_2	λ_3	λ_2

Table 8.45

First Tape

Discrete moment	0	1	2	3	4	5
ρ	ρ_2	ρ_1	ρ_1	ρ_3	ρ_2	ρ_1
$\varkappa$	$\varkappa_0$	$\varkappa_1$	$\varkappa_0$	$\varkappa_0$	$\varkappa_1$	$\varkappa_1$
λ	λ_1	λ_3	λ_2	λ_1	λ_2	λ_3

Table 8.46

Second Tape

Discrete moment	0	1	2
ρ	ρ_1	ρ_3	ρ_2
$\varkappa$	$\varkappa_0$	$\varkappa_0$	$\varkappa_1$
λ	λ_2	λ_1	λ_2

Table 8.47

Third Tape

Discrete moment	0	1	2	3
ρ	ρ_2	ρ_3	ρ_1	ρ_2
$\varkappa$	$\varkappa_0$	$\varkappa_1$	$\varkappa_2$	$\varkappa_1$
λ	λ_1	λ_2	λ_3	λ_2

Table 8.48

Basic Table of the Automaton

$\varkappa$ \ ρ	ρ_1	ρ_2	ρ_3
$\varkappa_0$	$\varkappa_0$	$\varkappa_1$	$\varkappa_1$
$\varkappa_1$	$\varkappa_0$	$\varkappa_1$	$\varkappa_2$
$\varkappa_2$	$\varkappa_1$	—	—

Table 8.49

Converter Table

$\varkappa$ \ ρ	ρ_1	ρ_2	ρ_3
$\varkappa_0$	λ_2	λ_1	λ_1
$\varkappa_1$	λ_3	λ_2	λ_2
$\varkappa_2$	λ_3	—	—

The same algorithm will also solve the less stringently specified problem where we are given a finite number of tapes of finite length and it is required to synthesize an s-machine realizing these tapes, but where the tapes need not all have the same initial state. In this case there is no need for checking whether the specification is contradictory, and the initial correct entries may be written immediately by appropriately selecting the initial states for each tape.

The key concept involved in our algorithm is the maximum use of states already present on the tapes, whereby new states are added only when absolutely necessary. This design procedure leads to a relatively economical machine. However, it does not, in general, give a minimal s-machine. This is because it may prove convenient to introduce a new state $\varkappa_{s+1}$, even though an already-existing state is suitable, if that will reduce the number of states in succeeding stages of synthesis.

So far, we have assumed that the number of pairs of input and output sequences is finite, and that all of these are enumerated in the specification. Now we shall discuss the general synthesis problem, where we do not assume that the number of the given correspondences between the input and output sequences is finite.

8.3. ALGORITHMIC UNSOLVABILITY OF THE PROBLEM OF RECOGNITION OF REPRESENTABILITY OF RECURSIVE EVENTS*

Let us assume that we have some description of the relations between the input and the output sequences which we want to duplicate. These relations may be completely arbitrary as long as their specifications can be *effectively described.* An effectively described specification is one which allows anyone familiar with the description to find that unique output sequence which corresponds to any input sequence of the specification.

To find out whether there exist s-machines capable of providing the desired input-output relations, we first must formalize the effective description of such relations. To do this, we turn to recursive description, which is the only known means of formalizing that which we intuitively express by the phrase "all that can be effectively specified by a human language."

*Readers not familiar with the theory of algorithms and recursive functions should read Chapter 12 prior to this section.

One method of describing input-output relations is based on representability of events, a concept we encountered in Chapter 7. Indeed, instead of specifying separately the output corresponding to each input sequence, we can specify the set G_i of all input sequences causing the operation of a given output λ_i. If such sets $G_0, G_1, \ldots, G_l$ are specified for all outputs $\lambda_0, \lambda_1, \ldots, \lambda_l$, we have a unique input-output relation. For example, suppose that the input alphabet is $\{\rho_1, \rho_2, \rho_3, \rho_4\}$ and the output alphabet is $\{\lambda_1, \lambda_2, \lambda_3\}$, and suppose further that:

a) the output λ_1 is generated at instant p if during the preceding two instants $[(p-2)$ and $(p-1)]$ the input sequence contained ρ_1 followed by ρ_4;

b) the output λ_2 is generated if the conditions of a) are not met and if there is no input ρ_3 during the internal $(p-3)$ to p;

c) the output λ_3 is generated in all the other instances. If this is the case, we can readily write out the output generated at any input sequence.

With this method of specifying input-output relations, the machine synthesis problem may be formulated as follows: given the events $G_1, G_2, \ldots, G_l$, we require an s-machine representing the event G_i by generating an output λ_i from alphabet $\{\lambda_1, \lambda_2, \ldots, \lambda_l\}$. Then the formalization of an *effective specification of the relation between sequences*, reduces to the formalization *of an event which can be effectively defined*. This last concept again can only be formalized in terms of a recursive description. Thus, when we say that an event G is given, we shall mean that what is given is a recursive description of the input sequence set G_i.

Let us agree that *an event G_i is recursive if the set G_i of input sequences is recursive*.*

We have already proved (Chapter 7) that only regular events are representable in an s-machine; we have also proved that irregular events do exist. Therefore, the problem of recognizing whether there exists an s-machine realizing some specific and effectively specified input-output relation becomes one of finding out whether some specific recursive event is regular; that is, we must find out if there exists an algorithm which, given any recursive event, can recognize whether this event is regular or not.

*Theorem 1***. *The problem of recognition of regularity of recursive events is algorithmically unsolvable*.

*For definition of a recursive set of sequences, see Chapter 12.

**A statement equivalent to this theorem was first advanced (without proof) in Sec. 5 of a paper by B. A. Trakhtenbrot [101] (see also [133] and [143]). (Footnote continued page 205)

Proof. Our proof will consist of formulating a problem narrower than that of recognition of representability of recursive events, and showing that even this narrow problem is algorithmically unsolvable. The broader theorem will then also have been proved.

Assume we are given a recursive function $\varphi(t)$ defined on the set of integers and assuming values from the finite set $\{0, 1, \ldots, r-1\}$. Then suppose we have an automaton A with an input alphabet $\{\rho_0, \rho_1, \ldots, \rho_{r-1}\}$. Of all its possible inputs, we shall note in particular the sequences

$$\rho_{\varphi(0)},$$
$$\rho_{\varphi(0)}\ \rho_{\varphi(1)},$$
$$\rho_{\varphi(0)}\ \rho_{\varphi(1)}\ \rho_{\varphi(2)},$$

where $\rho_{\varphi(i)}$ is a character from $\{\rho_i\}$, whose subscript coincides with the value of the recursive function $\varphi(t)$ at $t = i$.

Now consider an event S^{φ} consisting of the fact the input of the automaton contains one of the above sequences at that instant. In other words, an event S^{φ} occurs when, and only when, the subscripts of the inputs ρ coincide throughout (that is, at all instants $0, 1, 2, \ldots, p$) with the consecutive values $\varphi(0), \varphi(1), \varphi(2), \ldots, \varphi(p)$ of the given recursive function φ.

We shall say that the automaton A represents the recursive function φ if that automaton also represents the event S. But we already defined representation of events in Section 7.2 (p. 160). By analogy with that section, we shall say that an automaton represents a recursive function $\varphi(t)$ only if all of its states $\varkappa(p)$ belong to the allowed set M, and that these states can belong to M if, and only if, the subscripts of all inputs between $t = 0$ and $t = p$ are consecutive values of function $\varphi(t)$.

There exist recursive functions that are *a priori* known to be representable (for example, any periodic function is representable, since here the event S^{φ} is regular), as well as those that are *a priori* known to be unrepresentable (for example, the function $\varphi(t)$ that becomes 1 at $t = n^2$ and is zero in all the other instances). But in other cases, we are faced with the problem of recognizing the representability (or lack of it) of recursive functions. It can easily be seen that

Theorem 1 could be considered a direct result of Rice's theorem [103], if the class of recursive events were regarded as a class of events "generated" by all the possible recursive functions. However, one can also have recursive events "generated" by primitive recursive functions, and this case needs special treatment. Our proof, in addition to being general, is also completely applicable to events generated exclusively by primitive recursive functions.

this problem is a special case of our overall problem of recognition of recursive events.

In accordance with the theorem proved in Section 7.6, a recursive event S^{φ} is regular if, and only if, the recursive function $\varphi(t)$ is ultimately periodic. But since we know* that the problem of recognizing whether a given recursive function is ultimately periodic is algorithmically unsolvable, the same is true of the problem of recognition of regularity of event S^{φ}, and is all the more true of the broader problem of regularity of recursive events. This proves the Theorem 1.

Thus there is no algorithm capable of deciding whether a given recursive event is regular or not. The problem must be handled piecemeal, resorting in each particular instance to a ''creative'' (as opposed to a ''mechanical,'' that is, algorithmic) solution. Assume, however, that we are always able to separate out, in one way or another, the recursive events which are regular. Then, on the face of it, it would appear that we could design an algorithm for synthesizing automata representing those recursive events which are regular. However, it turns out that even this problem does not lend itself to a generalized solution. This is stated in another theorem of Trakhtenbrot, which we shall cite without proof.

Theorem 2. The problem of synthesis of an automaton representing an event from the set of all recursive events that are regular is algorithmically unsolvable.

The above two theorems lead to a very important conclusion: unless the allowable methods of specifying the desired machine (that is, the language describing the specification) is restricted in some way, any attempt to find an algorithmic method for synthesizing this s-machine will be meaningless. More than that, unless the language is restricted, any attempt to find a procedure for answering the mere question whether a machine realizing this specification exists at all will be doomed to failure. Fortunately, however, the language can be so restricted that any specification expressed in it will be *a priori* realizable by an s-machine. In this way, the recognition problem is completely avoided, and one needs to worry only about the synthesis problem.

One such restricted language is that of regular expressions, where the specifications are always written in terms of regular events. It is *a priori* known that there exists an algorithm for the synthesis of an s-machine specified in this restricted language. This existence follows from the reasoning employed in the proof of Kleene's first theorem (Chapter 7). A similar algorithm, again written in the

*See, for example, [142].

language of regular expressions but more convenient and yielding fewer states, is shown in Section 8.4. And B.A. Trakhtenbrot [101, 102] devised a predicate language also suitable for writing specifications which are *a priori* known to be realizable in some s-machine and for which there exists a synthesis algorithm.

However, the practical use of such languages merely shifts all the difficulties associated with the synthesis phase to the initial design phase, where the specifications are written. Indeed, the advantages inherent in these restricted languages are fully realized only if there are no intermediate translation steps, that is, if the specification is from the outset formulated in the appropriate language. Therefore, the designer issuing the specification must "think" in that language, that is, have the ability to express himself directly in it. However, in practice, the first definition of the required s-machine is inevitably expressed in words. This verbal definition must then be translated into the language of regular expressions. And one cannot accomplish this translation unless one knows beforehand that the specification is expressible in the language of regular expression. We are thus again trapped in a vicious circle.

A language suitable for specification and the subsequent synthesis must, therefore, satisfy the following three requirements:

1) Those verbal descriptions which are natural and frequently encountered must be easily translatable into this language.

2) The language must be so broad that those natural and frequently encountered verbal descriptions which are not realizable by an s-machine could also be translated into it.

3) Both the recognition and the synthesis problems must be algorithmically solvable for all the specifications written in this language.

So far, there are no languages satisfying all these conditions.

In the next section we shall consider a synthesis algorithm for the relatively easy case where the specification is given in the language of regular formulas, and the recognition problem therefore does not arise.

8.4. SYNTHESIS OF FINITE AUTOMATA AND SEQUENTIAL MACHINES IN THE LANGUAGE OF REGULAR EXPRESSIONS

Assume now that we are given one or more regular expressions (see Section 7.3), and it is required to synthesize an s-machine representing the input events specified by these expressions by generating the appropriate output symbols.* The problem then reduces to the synthesis of a finite automaton representing each of these events by an appropriate set of states.

Actually, this problem was already solved in Chapter 7, where Kleene's first theorem was effectively proved, that is, the proof of the theorem contained a method for constructing an automaton representing any event specified by a regular expression. If more than one regular expression is given, we can construct an automaton representing each of them separately, and then feed the outputs into the input of a common converter. However, we are confronted here with a situation similar to that already encountered in Section 8.2: we know a solution for the problem, but we are not content with it because the least number of states k in the resulting machine is too large for subsequent minimization.

We shall now present a method which does not suffer from this disadvantage, and which is an adaptation of a procedure proposed by V.M. Glushkov [252]. To begin with, assume we have one regular expression. We shall write it in a form somewhat different from that of Chapter 7. Thus in forming regular expression of Chapter 7, we started with finite segments of input tapes (that is, finite sequences of inputs ρ_i), which we then denoted by a, b, c, ... Now we shall start the inputs ρ_i themselves, that is, we shall employ only input sequences of length 1.

A regular expression consisting of finite sequences a, b, c, ... may be written in the form of a product. For example, the sequence $a=\rho_1\rho_5\rho_3\rho_4$ corresponds to the expression

$$R=\{[(\rho_1\cdot\rho_5)\cdot\rho_3]\cdot\rho_4\}.$$

A regular expression consisting of a, b, c, ... thus immediately yields the corresponding regular expression consisting of symbols ρ_i. For example, when $a=\rho_1$, $b=\rho_2\cdot\rho_3$ and $c=\rho_1\cdot\rho_2$, the regular expression of Chapter 7

$$R=\{[(b\vee c)^*]\vee(a\cdot c)\},$$

becomes

$$R=\{(\,[(\rho_2\cdot\rho_3)\vee(\rho_1\cdot\rho_2)]^*)\vee[\rho_1\cdot(\rho_1\cdot\rho_2)]\}.$$

It is obvious that the depth of this new regular expression may be much greater than that of the starting one.

*We are not concerned here with the criteria for the selection of these regular expressions.

The next step in our synthesis is representation of our regular expression in the form of a graph. To start with we adopt the following convention for expressions of depth 1:

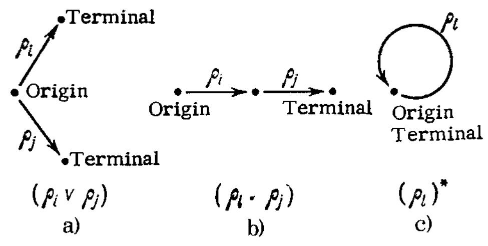

Fig. 8.1.

Thus, a disjunction$(\rho_i \vee \rho_j)$is shown (Fig. 8.1,a) by two arrows originating from a point and labeled ρ_i and ρ_j, respectively. This graph has one origin and two terminals. A product$(\rho_i \cdot \rho_j)$is shown (Fig. 8.1,b) by two respectively labeled arrows connected in series. This graph has one origin and one terminal. The iteration $(\rho_i)^*$ is shown (Fig. 8.1,c) by an appropriately labeled arrow closing upon itself. The origin of this graph is also its terminal.

In exactly the same manner, we define the operations of graphs of regular expressions R_1 and R_2 of depth $\geqslant 1$. Each such graph has one origin and at least one terminal (the origin and the terminal may also coincide, as in Fig. 8.1,c). The graph of the expression $(R_1 \vee R_2)$ is obtained by combining the origins of graphs for R_1 and R_2. The resulting graph has one origin and as many terminals as there are in the graph of R_1 plus the graph of R_2. The graph of $(R_1 \cdot R_2)$ is obtained by connecting all the terminals of the graph of R_1 with the origin of R_2 (so that the arrows in R_1 now point to the origin of R_2). The origin of the graph of $(R_1 \cdot R_2)$ then coincides with that of the graph of R_1, while the terminals are all those of the graph of R_2. The graph of $(R_1)^*$ is obtained from the graph of R_1 by joining all its terminals to its origin. The origin of this graph (which also is the origin of R_1) is thus also its terminal.

We shall now show a few examples of graphs of regular expressions.

The regular expression

$$R = \{\rho_1 \cdot [(\rho_2 \vee \rho_3)^*]\}$$

has the graph of Fig. 8.2 (which also shows the intermediate graphs).

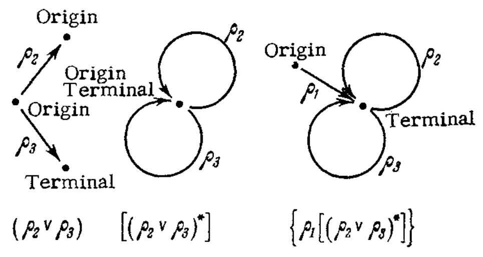

Fig. 8.2.

The regular expression

$$R = \{([(\rho_2 \cdot \rho_3) \vee (\rho_1 \cdot \rho_2)]^*) \cdot [\rho_1 \cdot (\rho_1 \cdot \rho_2)]\}$$

has a graph of Fig. 8.3.

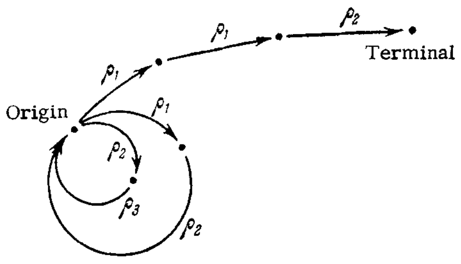

Fig. 8.3.

Let us stipulate that a graph depicting a regular expression must satisfy the following conditions: any path from the origin to a terminal must define a sequence whose input to the automaton signals the event specified by the given regular expression R; and, conversely, the graph must contain a path from the origin to one of the terminals for any input sequence signaling the occurrence of the regular event.

The graphs of Figs. 8.2 and 8.3 do satisfy these conditions. There are, however, regular expressions with graphs not conforming to these requirements. For example, the graph of

$$R = \{([\rho_1 \cdot (\rho_2)^*] \vee \rho_2) \cdot \rho_3\}$$

(see Fig. 8.4) contains "false paths." Here the path indicating an input sequence $\rho_2\rho_2\rho_3$ (heavy line) also corresponds to the expression

$$\{[\rho_2 \cdot (\rho_2)^*] \cdot \rho_3\}$$

that is, it does not signal the occurrence of the required event R. Again, Fig. 8.5 shows the graph of the regular expression

$$R = \{[(\rho_2)^* \cdot \rho_3] \vee \rho_1\}.$$

But the heavy-line path does not signal the occurrence of event R. Finally, consider the regular expression

$$R = \{(\rho_1 [(\rho_2)^* \cdot (\rho_3)^*]) \cdot \rho_4\}$$

whose graph is shown in Fig. 8.6. The graph contains a path $\rho_1 \rho_3 \rho_2 \rho_4$ which does not correspond to any specified event.

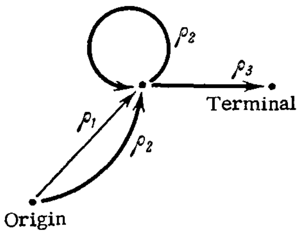

Fig. 8.4.

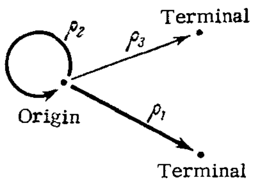

Fig. 8.5.

These three examples describe the three ways in which false paths may be generated. Thus, false path may result from the following operations:

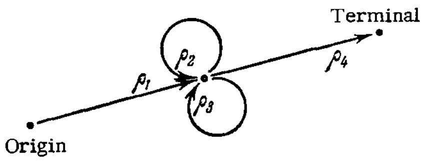

Fig. 8.6.

1) Multiplication by a disjunctive expression in which at least one of the disjunctive terms terminates in an iteration

$$[(\,[A \cdot (B)^*] \vee C) \cdot D];$$

2) Disjunction, in which at least one of the disjunctive terms starts with an iteration

$$\{[(A)^* \cdot B] \vee C\};$$

3) Multiplication of two iterations

$$[(A)^* \cdot (B)^*].$$

To avoid "false paths," the rules for construction of graphs are amended in these three cases to include *empty arrows* not labeled

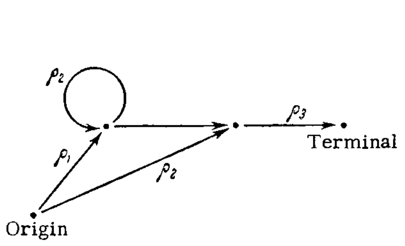

Fig. 8.7.

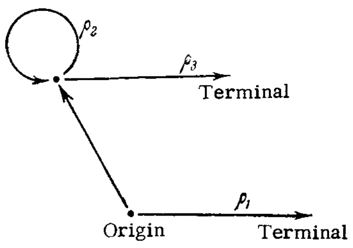

Fig. 8.8.

with symbols ρ. An empty arrow merely indicates the direction of movement along the graph and is "traversed" instantaneously, that is, it does not correspond to a discrete instant of operation of the automaton.

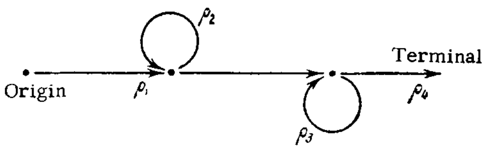

Fig. 8.9.

We can now correct the graphs of Figs. 8.4 - 8.6 by means of empty arrows, to give Figs. 8.7—8.9. In the first case, the empty arrow starts at the end of the iteration; in the second case, it is interposed between the common origin of the graph and the start of the iteration; in the third case, the empty arrow is interposed between the end of the first and the start of the second iteration. The corrected graphs still represent the respective regular expressions, but no longer contain false paths.

We shall demonstrate the synthesis of the automaton corresponding to the regular expression

$$\begin{aligned} R = \{[(\rho_1 \cdot \{[(\rho_2)^* \cdot \rho_3] \vee [\rho_1 \cdot (\rho_2)^*]\}) \cdot (\rho_3)^*] \vee \\ \vee ([(\rho_1 \cdot \rho_2)^*] \cdot \{[\rho_3 \vee (\rho_2 \cdot \rho_2)]^*\} \cdot \rho_1) \}. \end{aligned} \tag{8.1}$$

Its correct graph is shown in Fig. 8.10.

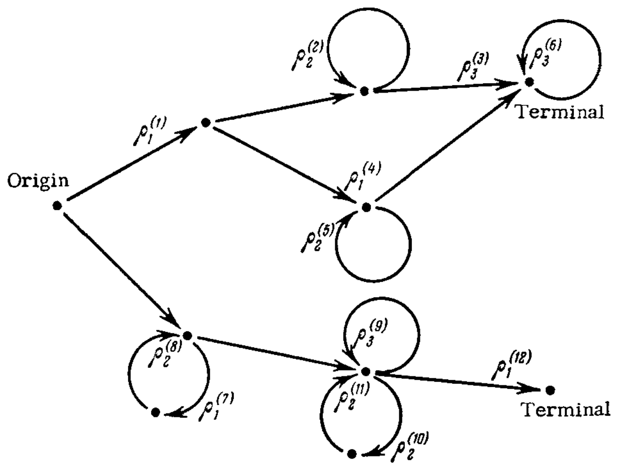

Fig. 8.10.

Now we number consecutively, from left to right, all the symbols ρ in this expression, entering the resulting ordinal numbers as superscripts (this will result in different superscripts on identical ρ_i's). We then obtain

$$R = \{[(\rho_1^{(1)} \cdot \{[(\rho_2^{(2)})^* \cdot \rho_3^{(3)}] \vee [\rho_1^{(4)} \cdot (\rho_2^{(5)})^*]\}) \cdot (\rho_3^{(6)})^*] \vee \\ \vee ([(\rho_1^{(7)} \cdot \rho_2^{(8)})^*] \cdot \{[\rho_3^{(9)} \vee (\rho_2^{(10)} \cdot \rho_2^{(11)})]^*\} \cdot \rho_1^{(12)})\}. \tag{8.2}$$

The same superscripts are then assigned to the corresponding labels ρ_i on the graph*, as shown in Fig. 8.10.

Now we write out at each node of the graph the superscripts of the ρ_i's of the arrows *converging upon it*, assigning the superscript 0 to the starting node. The number of the node which is the origin of an empty arrow is written at the node upon which that empty arrow converges. If two or more arrows with identical label superscripts converge on a single node, the superscript is written at that node only once. Figure 8.11 shows the graph of our example with all the numbers in place.

We now construct a table whose column headings correspond to the various ρ_i's of the regular expression. The heading of the first row is the symbol $*$, which is also entered in the entire row (Table 8.50). The heading of the second row is 0, and each column ρ_i contains the superscripts of all the arrows labeled ρ_i and originating from the nodes whose description includes 0. We thus obtain Table 8.51.

*With this method, it is usually convenient to number the symbols of the regular expression first, and only then construct the graph.

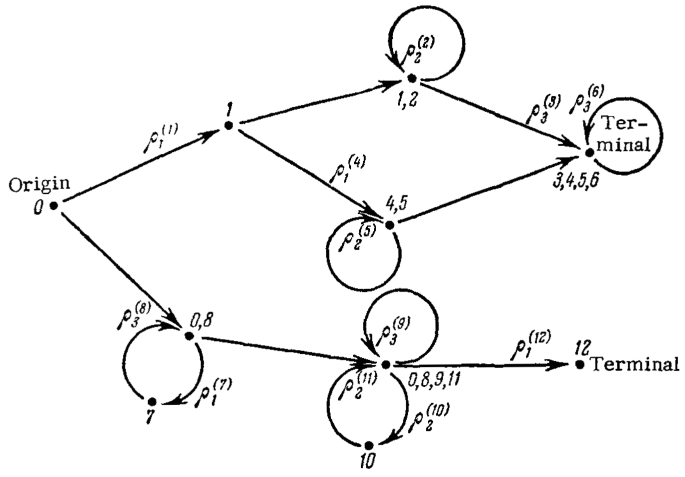

Fig. 8.11.

Table 8.50

ρ / Superscript	ρ_1	ρ_2	ρ_3
*	*	*	*

Table 8.51

ρ / Superscript	ρ_1	ρ_2	ρ_3
*	*	*	*
0	1, 7, 12	10	9

Now we add as many rows as there are different entries in row 0, with these entries becoming the headings of the new rows. We then obtain Table 8.52. The columns ρ_i in each new row are now filled in a manner similar to that used in filling row 0. For example, the intersection of row 1, 7, 12, and column ρ_3 contains the superscripts k of the labels ρ_3^k of all the arrows originating from the nodes whose description includes 1, 7, or 12. If there are no such arrows, then we enter the symbol *. As a result of this procedure, we get Table 8.53. After this, we add to Table 8.53 as many rows as there are *new* entries in the three rows just completed, and fill in the columns in the manner just demonstrated. The table will be completed in a finite number of steps, since the number of different combinations of superscripts k is finite. If N is the number of characters in the given regular expression, then we have a total of $N + 1$ superscripts and no more than 2^{N+1} different superscript combinations, that is, different table

Table 8.52

Superscript \ ρ	ρ_1	ρ_2	ρ_3
*	*	*	*
0	1, 7, 12	10	9
1, 7, 12			
10			
9			

Table 8.53

Superscript \ ρ	ρ_1	ρ_2	ρ_3
*	*	*	*
0	1, 7, 12	10	9
1, 7, 12	4	2	3, 8
10	*	11	*
9	12	10	9

entries. The number of the rows in the table cannot, therefore, exceed 2^{N+1}.

Finally, we check off (on the left margin) all those rows whose headings contain a superscript appearing in the description of a terminal node of the graph, and obtain Table 8.54.

The next step is to code the row headings of the table by means of consecutive symbols $\varkappa_0, \varkappa_1, \ldots$; we code the table entries accordingly.

Table 8.55 thus constructed is the basic table of a finite automaton which, starting from an initial state $\varkappa_1$, represents the events defined by the regular expression (8.1) by the set of checked-off states (states $\varkappa_2, \varkappa_5, \varkappa_7, \varkappa_9, \varkappa_{10}, \varkappa_{11}, \varkappa_{12}, \varkappa_{13}, \varkappa_{14}$). To convince ourselves of this, let our automaton be in an initial state $\varkappa_1$ and let the input sequence be $\rho_1\rho_3\rho_1$. The automaton will then go to state $\varkappa_{13}$. The symbol $\varkappa_{13}$ (compare Tables 8.54 and 8.55) is the code for the superscript set 7, 12. But then it follows from the very procedure for construction of Table 8.54 that the graph of Fig. 8.11 contains a path $\rho_1^{i_1}\rho_3^{i_2}\rho_1^{i_3}$ (starting at the origin and possibly including some empty arrows) such that $\rho_1^{i_3}$ is equivalent to ρ_1^7 or ρ_1^{12}, that is, $i_3 = 7$, or $i_3 = 12$. Now, does the sequence $\rho_1\rho_3\rho_1$ signal the occurrence of the specified event? We can reformulate this question as follows: is there a graph path $\rho_1\rho_3\rho_1$ from the origin to one of the terminals? Table 8.54 and Fig. 8.11 provide the answer: since path $\rho_1\rho_3\rho_1$ can terminate in ρ_1^{12} and an arrow so labeled does lead to a terminal node, this path does exist. Thus, whenever an input sequence resets the automaton into a checked-off state, this means that the corresponding graph path leads

Table 8.54

	Superscript \ ρ	ρ_1	ρ_2	ρ_3
	*	*	*	*
	0	1, 7, 12	10	9
V	1, 7, 12	4	2	3,8
	10	*	11	*
	9	12	10	9
V	4	*	5	6
	2	*	2	3
V	3,8	7, 12	10	6,9
	11	12	10	9
V	12	*	*	*
V	5	*	5	6
V	6	*	*	6
V	3	*	*	6
V	7, 12	*	*	8
V	6, 9	12	10	6,9
	8	7, 12	10	9

Table 8.55

	$\varkappa$ \ ρ	ρ_1	ρ_2	ρ_3
	$\varkappa_0$	$\varkappa_0$	$\varkappa_0$	$\varkappa_0$
	$\varkappa_1$	$\varkappa_2$	$\varkappa_3$	$\varkappa_4$
V	$\varkappa_2$	$\varkappa_5$	$\varkappa_6$	$\varkappa_7$
	$\varkappa_3$	$\varkappa_0$	$\varkappa_8$	$\varkappa_0$
	$\varkappa_4$	$\varkappa_9$	$\varkappa_3$	$\varkappa_4$
V	$\varkappa_5$	$\varkappa_0$	$\varkappa_{10}$	$\varkappa_{11}$
	$\varkappa_6$	$\varkappa_0$	$\varkappa_6$	$\varkappa_{12}$
V	$\varkappa_7$	$\varkappa_{13}$	$\varkappa_3$	$\varkappa_{14}$
	$\varkappa_8$	$\varkappa_9$	$\varkappa_3$	$\varkappa_4$
V	$\varkappa_9$	$\varkappa_0$	$\varkappa_0$	$\varkappa_0$
V	$\varkappa_{10}$	$\varkappa_0$	$\varkappa_{10}$	$\varkappa_{11}$
V	$\varkappa_{11}$	$\varkappa_0$	$\varkappa_0$	$\varkappa_{11}$
V	$\varkappa_{12}$	$\varkappa_0$	$\varkappa_0$	$\varkappa_{11}$
V	$\varkappa_{13}$	$\varkappa_0$	$\varkappa_0$	$\varkappa_{15}$
V	$\varkappa_{14}$	$\varkappa_9$	$\varkappa_3$	$\varkappa_{14}$
	$\varkappa_{15}$	$\varkappa_{13}$	$\varkappa_3$	$\varkappa_4$

to a terminal node, that is, this input sequence corresponds to a path from the origin to a terminal node. But this, in turn, means that the specified event has occurred. Therefore, our automaton represents this specified event. Whenever the input sequence is not the initial segment of any sequence specifying an event (and therefore no event will occur), our automaton is reset into state $\varkappa_0$ and stays in it. In Table 8.55, such a situation arises with input sequences $\rho_1\rho_1\rho_1$, $\rho_1\rho_3\rho_2\rho_1$, and so on.

If more than one regular event (R_1, R_2, ..., R_m) is specified, our method is modified as follows: The symbols in the regular expression are numbered consecutively (that is, the ρ_i's in R_1 are assigned superscripts 1, 2, ..., m, those in R_2 are assigned superscripts $m + 1$, $m + 2 \ldots n$, and so on). A separate graph is then constructed for each expression. The superscript 0 is assigned to the origins of all the graphs. A common table is constructed, so that an intersection of a row and a column may contain superscripts from several graphs. Then the rows containing the superscripts marking terminal nodes of all graphs are checked off (that is, the sets of symbols representing each of the events $R_1, R_2, \ldots, R_m$ are determined). Then one designs an output converter which places an appropriate symbol λ with each of these sets of states.

Example. Given three events

$$R_1 = [\rho_1 \cdot (\rho_2 \vee \rho_1)],$$
$$R_2 = [\rho_2 \cdot (\rho_3)^*],$$
$$R_3 = \{\rho_1 \cdot [(\rho_2)^* \vee \rho_3]\}.$$

The corresponding graph is shown in Fig. 8.12, and the above-described algorithm produces Tables 8.56 and 8.57. However, now we do not use check marks, but label the states representing the events R_1, R_2, and R_3 with symbols λ_1, λ_2, and λ_3, respectively. State $\varkappa_5$ is labeled with two symbols (λ_1, λ_3) because events R_1 and R_3 contain a common sequence ($\rho_1 \cdot \rho_2$) leading to $\varkappa_5$. Therefore, we either identify λ_1 with λ_3 (that is, fail to distinguish between events R_1 and R_3), or we must label $\varkappa_5$ with a new symbol λ_4.

We could have synthesized our automaton directly from the regular expression, without using a graph. In fact, this is the procedure used by the author of our method, V.M. Glushkov, and it may prove to be more convenient in those cases where the regular expressions R_i yield complicated, cumbersome graphs. That procedure is, however, not as easy to visualize as that employed in this book.

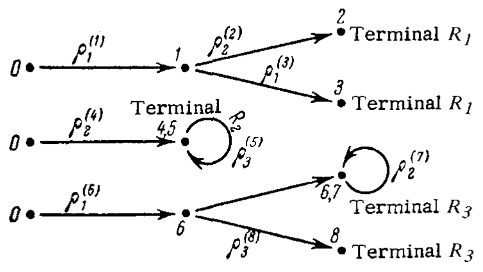

Fig. 8.12.

The obvious problem which arises in connection with the Glushkov method is that of *a priori* estimation of the number of states in the final automaton. We shall present (without proof) an estimate for the

Table 8.56

	ρ	ρ_1	ρ_2	ρ_3
	*	*	*	*
	0	1,6	4	*
λ_3	1,6	3	2,7	8
λ_2	4	*	*	5
λ_1	3	*	*	*
λ_1, λ_3	2,7	*	7	*
λ_3	8	*	*	*
λ_2	5	*	*	5
λ_3	7	*	7	*

Table 8.57

	$\varkappa \backslash \rho$	ρ_1	ρ_2	ρ_3
	$\varkappa_0$	$\varkappa_0$	$\varkappa_0$	$\varkappa_0$
	$\varkappa_1$	$\varkappa_2$	$\varkappa_3$	$\varkappa_0$
λ_3	$\varkappa_2$	$\varkappa_4$	$\varkappa_5$	$\varkappa_6$
λ_2	$\varkappa_3$	$\varkappa_0$	$\varkappa_0$	$\varkappa_7$
λ_1	$\varkappa_4$	$\varkappa_0$	$\varkappa_0$	$\varkappa_0$
λ_1, λ_3	$\varkappa_5$	$\varkappa_0$	$\varkappa_8$	$\varkappa_0$
λ_3	$\varkappa_6$	$\varkappa_0$	$\varkappa_0$	$\varkappa_0$
λ_2	$\varkappa_7$	$\varkappa_0$	$\varkappa_0$	$\varkappa_7$
λ_3	$\varkappa_8$	$\varkappa_0$	$\varkappa_8$	$\varkappa_0$

case where the automaton represents specific events* (specific events are a subclass of regular events).

Let specific events $ER_1, ER_2, \ldots, ER_m$ be given. Now consider the formula

$$S = R_1 \vee R_2 \vee \ldots \vee R_m.$$

This formula contains no iterations (since the events are specific) and, therefore, graph S has no loops (that is, no path crosses the same node twice). Formula S can be transformed to a form S' such that: a) the graph of S' will be a tree, that is, only one arrow will terminate in each node, and, therefore, different paths will always lead to different nodes; and b) if two or more arrows originate at a single node, they all will be labeled with different characters of the input alphabet. Formula S' will be equivalent to S, but may differ from it in the number of characters it contains. Let us denote the number of characters in S' as N'. Then one can synthesize an automaton representing the system of events $ER_1, ER_2, \ldots, ER_m$. The number of states in this automaton will not exceed $N' + 1$. In practice, however, this method yields automata with a much smaller number of states than $N' + 1$.

*The upper limit for the number of states in the case of an arbitrary regular event was estimated by Glushkov in [29].