LOGIC, AUTOMATA, AND ALGORITHMS

Volume 79

Mark A. Aiserman &
Leonid A. Gusev

Logic, Automata, and Algorithms

Mark A. Aiserman
Leonid A. Gusev
Lev I. Rozonoer
Irina M. Smirnova
Aleksey A. Tal'

Institute of Automation and Remote Control
Academy of Sciences of the USSR, Moscow

Translated by Scripta Technica, Inc.

TRANSLATION EDITOR

George M. Kranc
City University of New York
New York, New York

ACADEMIC PRESS 1971 NEW YORK AND LONDON

ACADEMIC PRESS, INC.
111 Fifth Avenue, New York, New York 10003

United Kingdom Edition published by
ACADEMIC PRESS, INC. (LONDON) LTD.
Berkeley Square House, London W1X 6BA

LIBRARY OF CONGRESS CATALOG CARD NUMBER: 72-153664

Originally published as:
"Logika, Avtomaty, Algoritmy" by Gosud.
Iz-vo Fiziko-Matem. Literatury (State Press
for Physical and Mathematical Publications),
Moscow, 1963

PRINTED IN THE UNITED STATES OF AMERICA

Contents

Preface

This book deals with the general theory of finite automata and sequential machines, a subject of great current theoretical and practical importance and one likely to have an even greater impact in the future.

In writing this text, we had in mind a wide audience. We naturally hoped it would be useful to specialists in switching or digital computer theory and design. Such persons are already familiar with the necessary mathematical techniques, that is, propositional calculus, general concepts of predicate calculus, and the fundamentals of the theory of algorithms (theory of recursive functions). For them, the book may serve as a reference on fundamentals. But our primary audience is the beginner whose mathematical training is confined to fundamentals of calculus, differential equations, and mathematical physics. Aside from engineering students, such readers may include specialists in automation, remote control and communications, that is, those branches of engineering where lack of fundamentals of mathematical logic and the theory of algorithms may preclude the solution of a variety of problems.

In addition, we would like to think that the potential beneficiaries may include the mathematician who is not a logician, as well as the physicist, physiologist and biologist interested in the applications of the theory of finite automata and sequential machines to idealized models, such as those of nets of nerves. Basically, however, the book is intended for engineers, which is why, in discussing some problems of logic and algorithmic theory, we preferred to forego mathematical rigor and concentrate on the clarity of exposition.

Thus, the objective of this book is to introduce the reader to this new field and familiarize him with the basic concepts and the ways in which particular problems are stated, as well as those solutions which have been obtained so far. In the presentation, our own results are intertwined with those obtained from the relevant literature.

Since this text is designed for a diversified audience, we could not organize it in a manner that would suit any special group. The

disposition of subject matter is thus a compromise between contending interests. In general, the material is arranged in order of increasing difficulty, and each reader should thus proceed according to his own needs and background. We would, however, like to offer several suggestions:

1. The reader who is completely unfamiliar with the subject but seeks detailed information should follow the sequence presented in the book.
2. The reader interested only in a general acquaintance with the subject should read the first seven chapters consecutively, followed by Chapter 12; after this he may glance through Chapter 13, and finish by reading Chapters 8, 9, 10, and 11.
3. The reader familiar with the fundamentals of mathematical logic and its technical applications is advised to begin with Chapter 3.
4. Finally, the mathematician interested in engineering applications may safely omit Chapters 1, 12, and 13.

Sections 2.5 and 8.4 deal with the special problems of minimizations of Boolean functions and the realization of finite automata, defined in the language of regular expressions; these sections (which go beyond the basic principles of the general theory of finite automata and sequential machines) were written, at the author's request, by V.D. Kazakov and O.P. Kuznetsov, respectively.

The authors would welcome all comments and suggestions.

TRANSLATOR'S NOTE

This translation of the original Russian edition contains problems, additions, and revisions prepared by the author for the English edition.

Introduction

"Finite automaton" and "sequential machines" are two traditional terms that are widely used to designate a very simple class of dynamical systems. The theory of this class evolved as a separate entity for the following two reasons:

1. These dynamical systems are frequently employed in technology, particularly in automatic and remote control and computer engineering (digital computers are a special case of this class). The needs of modern technology have therefore prompted an intensive study of the general relationships governing this class, in order to develop methods of analysis and of optimal synthesis of these dynamical systems.

2. The continuing progress in science and technology, particularly in computers, increasingly poses questions such as: What can a machine "do" and what is it incapable of "doing"? Could a machine perform any algorithm? In principle, could a machine do something more than merely execute an algorithm? To what extent is a machine capable or incapable of performing functions characteristic of a human brain? All attempts at exact formulation of these questions, let alone finding the answers, hinge upon our definition of the term "machine." As of now, it is impossible to solve these problems in terms of a very broad class of dynamical systems. If, however, we define a machine as a restricted class of such systems—that known as "finite automata" and "sequential machines"—then the questions make sense. They can be exactly formulated and, in some cases, answered.

There is another reason, peculiar to our present state of knowledge, which helps maintain interest in systems of this class. The brain consists of a very large number of nerve cells, or neurons. By idealizing their properties to some extent, we can construct a mathematical model of the brain—one that is valid, of course, only within the limits of this idealization. This model is also a dynamical system of the type we shall consider. Our expanding knowledge of neurons and of the brain as a whole has shown that the above idealization is inadequate and that more complex models are desirable. Nevertheless, the fact remains that within this idealized frame-

work, which is quite acceptable at our particular stage of knowledge, both the human brain and a general-purpose digital computer can be regarded as belonging to the same comparatively simple class of dynamical systems. It is this fact that lends interest to the study of finite automata and sequential machines.

1

Elements of Mathematical Logic

1.1. INTRODUCTORY NOTES

Mathematical (symbolic) logic traces its origins to the so-called traditional formal logic, from which it emerged in response to a desire to formalize certain aspects of intellectual activity. This desire continued to influence its subsequent growth as an independent science, when it addressed itself to the task of providing the logical foundations of mathematics by tackling problems of consistency and completeness of axiomatic systems underlying this science, the problem of determining all the inferences derivable from these axioms, as well as a variety of similar questions. Eventually mathematical logic grew into a powerful research tool, but its use continued to be restricted to the domain of pure theory, even though there were men who recognized its potential in the field of applied science (as long ago as 1910 Paul Ehrenfest pointed out the possibility of using the constructs of mathematical logic to describe the operation of practical systems such as switching circuits). Be that as it may, it was only in the thirties that the engineering application of mathematical logic came into its own. It was during that time that V.I. Shestakov [111, 112] and C.E. Shannon [231] worked on the application of mathematical logic to switching networks and led the way for M.A. Gavrilov [21] and the independent theory of relay switching. Before long mathematical logic penetrated even deeper into the applied sciences. It was found that not only relay switching networks but also many other discrete-action systems were susceptible to description by its techniques.

Thus mathematical logic became an accepted tool in the development and design of a great variety of engineering systems, while at the same time maintaining its extreme importance in theoretical research. Its applied aspect proved especially valuable in recent years, in connection with the research into the general laws of control which govern both technology and Nature.

Since there are two aspects of mathematical logic—the theoretical and the applied—the subject can be developed in two distinct ways. In accordance with our main objective, we shall confine ourselves to the applied aspect, with the further restriction that we shall now discuss only these elements of logic which are needed for an understanding of later sections.

1.2. BASIC CONCEPTS

In discussing logic, we shall experience time and again the importance of a fundamental mathematical concept—the *functional relationship*. In its most general form this concept is associated with the idea of existence of two sets and of mapping of one set onto the other. Suppose we have sets X and Y consisting of elements x and y, respectively, that is,

$$X = \{x\}, \quad Y = \{y\}.$$

If, by virtue of some condition, each element x belonging to set X (this is written as $x \in X$) is matched with a specific element y of set Y ($y \in Y$), then the matching condition is said to define y as a function of x, or, alternatively, one says that set X maps into set Y. The function $y = y(x)$ is also said to be defined on the set X (called the domain of the function) and to have values in set Y (the range of the function); x is called the *independent variable or argument*, and y is called the *function*.

Every specific functional relationship is determined, on the one hand, by the characteristics of sets X and Y and, on the other hand, by the nature of elements x and y in these two sets.

Let us consider some basic characteristics of sets. A set is classed as either *finite* or *infinite*, depending upon the number of elements constituting it. For example, the set of letters of the alphabet is finite; the set of molecules in a finite body is also finite; but sets consisting of all positive integers, or of all rational numbers, or of all real numbers are infinite. The set of all points on a line segment and the set of all points in a plane figure are also infinite.

Sets may be compared according to their cardinality. Two sets are said to have the same cardinality if a one-to-one correspondence can be established between their elements. The concept of cardinality of a set allows us to distinguish two important classes of infinite sets. These are the *countable** and the *continuum* sets.

*Also called denumerable.

Countable sets are sets that have the cardinality of the set of all natural numbers, and continuum sets are sets that have the cardinality of the set of all real numbers.

In particular, the set of all even integers is countable, since the elements of this set can be easily placed in a one-to-one correspondence with the elements of the set of natural numbers. Indeed, by arranging the even integers and the natural numbers in ascending order, we can establish the following one-to-one correspondence between the elements of these two sets:

$$2,\ 4,\ 6,\ \ldots,\ 2n,\ \ldots$$

$$1,\ 2,\ 3,\ \ldots,\ n,\ \ldots$$

The set of all algebraic numbers, the set of all rational numbers, and so on, are also countable.

The continuum sets include the set of all irrational numbers, the set of all points in a line segment, the set of all points on a plane figure, and many others.

In some cases, comparison of infinite sets in terms of their cardinality leads to statements that may sound quite paradoxical. For instance, it would seem strange, at a first glance, that the set of points in a segment (AB in Fig. 1.1) and the set of points in a section of the same segment (AC in Fig. 1.1) should have the same cardinality. This, however, may easily be proven with the help of Fig. 1.1. Here, each point M in segment AB may be connected by a ray to an origin O; this ray intersects segment AC at a point M', which is seen to be in one-to-one correspondence with point M of segment AB, showing that our two sets do indeed have the same cardinality. Similarly, it may be demonstrated that the set of points in a plane figure, or even in a three-dimensional body, has the same cardinality as the set of points in a line segment, namely, that of the continuum.

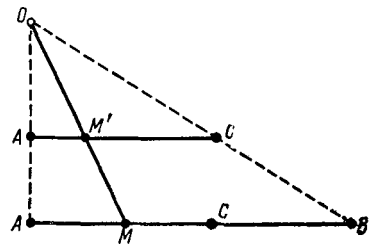


Fig. 1.1.

Let us now return to functional relationships. As already stated, such a relationship is specified by the nature of the elements in the sets on which it is defined, and by the characteristic properties of these sets. If a function is defined on the set X of all real numbers x and assumes values from a set Y which also consists of all real numbers y, then we have a real function y of a single real variable

x, or $y = y(x)$. If, however, the function assumes values from the same set of real numbers y, but each element of the set $Z = \{z\}$ on which it is defined is a sequence of n real numbers $x_1, x_2, \ldots, x_n$, then we are no longer dealing with a real function of a single real variable, but with a real function y of n real variables $x_1, x_2, \ldots, x_n$, that is, $y = y(x_1, x_2, \ldots, x_n)$.

The above functions are based on the set of real numbers, and it is this characteristic that unites them into a single class. The distinguishing feature of this class of functions is that both the values assumed by the function and the arguments of this function are defined on continuum sets.

The basic characteristic of functions of mathematical logic is that both their domain and their range (that is, the sets which participate in the mapping) consist of elements that, in general, have no connection with any defined quantities whatsoever. We are thus saying that we cannot distinguish between the elements of these sets by any other means than assigning to them symbols of some kind, for example, numerals.

The list of all symbols describing the elements of a given set is called the *alphabet* of this set; an undefined symbol, which may represent any element of the set, is called a ***logical variable***. Each specific symbol is then one of the values which the logical variable can assume.

Thus we have seen that, in terms of the properties of the elements of the mapped sets, logical functions are functions of the most general type. Moreover, they assume values from finite sets. In this they differ from many other functions (for example, functions of real variables, which are, in general, defined on continuum sets).

As an example, consider two sets. Set $X = \{x\}$ consists of all the *different* white keys of the piano. Let us denote these keys, from left to right, by symbols $x_1, x_2, \ldots, x_{50}$; the list of these symbols is alphabet of set $X = \{x_1, x_2, \ldots, x_{50}\}$. Set $Y = \{y\}$ consists of the seven different notes contained in an octave, and its alphabet is $\{y_1, y_2, \ldots, y_7\}$, where the symbols $y_1, y_2, y_3, y_4, y_5, y_6$, and y_7 denote the notes c, d, e, f, g, a, and b, respectively. In a well-tuned piano each symbol of the alphabet $\{x\}$ is in a one-to-one correspondence with a specific symbol of the alphabet $\{y\}$. This means that the variable y, which assumes the values $y_1, y_2, \ldots, y_7$, is a logical function of the independent variable x, which assumes the values $x_1, x_2, \ldots, x_{50}$. This function may be specified in several ways, for example, in the form of a table (see Table 1.1).

The first classification to which we may subject the functions of mathematical logic is that based on the number of different sets

involved in the mapping of a given function. If only one set is involved, so that the set is mapped into itself, the corresponding logical function is said to be *homogeneous*. A function involving mapping of one set onto a different set is said to be *heterogeneous*. For example, the logical function given in Table 1.2 is homogeneous, while that of Table 1.3 is heterogeneous.

Table 1.1

x	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	...	x_{44}	x_{45}	x_{46}	x_{47}	x_{48}	x_{49}	x_{50}
y	y_6	y_7	y_1	y_2	y_3	y	y_5	y_6	y_7	y_1	...	y_7	y_1	y_2	y_3	y_4	y_5	y_6

We said before that the set from which a logical function takes its values is finite; and since any homogeneous logical function represents the mapping of some set onto itself, it follows that the set constituting the range of a homogeneous logical function must be finite. The corresponding logical variable may be two-valued, three-valued, or, in general, m-valued.

Table 1.2

α_1	α_2	α_3	α_4
α_2	α_1	α_3	α_3

Table 1.3

α_1	α_2	α_3	α_4
β_2	β_1	β_1	β_3

Each of the values of the argument of a heterogeneous logical function is usually called an *object*, and the function itself is called a *predicate*. While the set of the argument values (the object set) may be infinite, the heterogeneous logical functions themselves—the predicates—may only be two-valued, three-valued, or, in general, m-valued (where m must be finite).

In the theory of real variables, we are accustomed to real functions of n real arguments. In the same way, the theory of logical variables admits of logical functions not only of one but also of n independent variables.

We shall divide functions of several variables into two classes. One of the classes shall include functions in which all the arguments, as well as the function itself, are logical variables assuming values

from the same set. Again, we shall call such functions "homogeneous."* Our second class shall comprise all those logical functions of several variables which do not belong to the first class; again, we shall call such functions "heterogeneous."

As in the case of functions of one independent variable, the logical variables that are the arguments of the heterogeneous functions of several variables are called objects; the functions themselves are again called predicates.

Depending on the number of arguments in a given heterogeneous logical function, we have *one-place*, *two-place*, and, in general, *n-place predicates*. One-place predicates are sometimes called *properties*, while multiple-place predicates are called *relations*.

To illustrate these concepts and terminology, let us consider a few examples:

Suppose we examine the event: I shall meet a man whom I know. This event may or may not occur, depending on occurrence or nonoccurrence of the following elementary events constituting the composite event: One of the persons I shall meet will be someone I know, and this person shall also be a man. Here we have a homogeneous logical function with two arguments; it is homogeneous because both arguments and the function itself are events, that is, logical variables assuming values from the same binary set whose elements are "the event shall occur" and "the event shall not occur." By denoting one argument (event: meeting a person whom I know) by x_1, the other (event: meeting a man) by x_2, and the function (event: meeting a man whom I know) by y, we can represent their relationship in the form of Table 1.4. The characters 0 and 1 in the table are symbols corresponding to the elements "the event shall not occur" and "the event shall occur."

Table 1.4

x_1 \ x_2	0	1
0	0	0
1	0	1

In our previous example of the piano keys, the logical function was heterogeneous. There we had a seven-valued, one-place predicate whose object variable (the number of the key) assumed values from a fifty-element set.

The estimation of the truth value of a statement given by the algebraic expression

$$x_1 + x_2 > 10,$$

*The mathematicians who developed the theory of homogeneous logical functions worked with a set whose elements were called "true" and "false" propositions. For this reason this theory is referred to as "propositional calculus."

which is true for some numerical values of x_1 and x_2 and false for other, leads us to an example of a two-place, two-valued predicate; here we have two independent variables, and they assume values from a set of real numbers, which has the cardinality of the continuum.

Consider another example: it is no great trick to determine the day of the week corresponding to a certain date (day, month, year). The rules governing this problem constitute a heterogeneous logical function—a three-place, seven-valued predicate. The object variables in this case assume values from three sets: one of these contains 31 elements, another 12 elements, and the third a countable number of elements.

No single mathematical theory applicable to all the possible logical functions exists as yet. The theory which as of now has reached the highest state of development is that governing two-valued functions. This branch of mathematical logic (two-valued or binary logic) serves a dual function: on the one hand, it supports the entire edifice of mathematical logic; on the other hand, it is precisely this branch of the theory that is, at present, of the greatest applied value. The same, however, cannot be said about the theory of many-valued logic, which is still a long way from perfection. For this reason we shall not concern ourselves with it any further and shall proceed to the postulates of binary logic which includes the calculi of two-valued propositions and predicates.

1.3. PROPOSITIONAL CALCULUS

a) Definition of Logical Functions

We shall now discuss homogeneous binary logical functions

$$y = y(x_1, x_2, \ldots, x_n),$$

that is, functions in which all the independent variables $x_1, x_2, \ldots, x_n$, as well as the function y itself, assume values from the same binary set M. We shall denote the two elements of this set by symbols 0 and 1; these symbols shall then constitute the entire alphabet of all the logical variables which are arguments of these logical functions.

Now let us construct a table (see Table 1.5) of 2^n columns and n rows. The heading of each row shall be one of the n independent variables. The heading of each column will be a numeral from the set $0, 1, 2, \ldots, 2^n - 1$.

Next, let us fill each column with a sequence of symbols 0 and 1 such that this sequence, when read from bottom to top, shall form

Table 1.5

$r = 2^n$ columns

x \ k	0	1	2	3	4	5	6	7	...	2^n-2	2^n-1
x_1	0	1	0	1	0	1	0	1	...	0	1
x_2	0	0	1	1	0	0	1	1	...	1	1
x_3	0	0	0	0	1	1	1	1	...	1	1
...	...	...	...	...	...	...	...	...	...	...	...
x_n	0	0	0	0	0	0	0	0	...	1	1

n rows

the binary representation of the numeral in the heading of the column. The best way to complete such a table is as follows: We enter in the first row (row x_1) a string of pairs (01); in the second row (row x_2), a string of groups of four (0011); in the third row, a string of groups of eight (00001111), and so on. Now each column of the table shows one of the possible combinations of values which may be assumed by the n independent variables. Thus it may be said that each column corresponds to a point in an n-dimensional binary logical space (a space constructed on the basis of the two-element set M). The table as a whole (the aggregate of all the 2^n columns) is a complete description of the entire n-dimensional binary logical space which consists of $r = 2^n$ points; the numeral k heading a column is then a symbol denoting a point in this space.

In a more graphic representation of an n-dimensional binary logical space, 0 and 1 can be regarded as real numbers. Then the one-dimensional case may be represented in terms of two points on a real axis (Fig. 1.2). The two-dimensional case may be represented in terms of four vertices of a unit square (Fig. 1.3), and the three-dimensional case—by the vertices of a unit cube (Fig. 1.4). In general, the n-dimensional binary logical space may be represented in terms of the set of all the vertices of an n-dimensional unit cube.

Fig. 1.2.

To define a specific binary homogeneous logical function $y = y(x_1, x_2, \ldots, x_n)$ means to specify which of the two possible values

(0 or 1) will be assumed by the logical variable y at a point k of the corresponding binary n-dimensional logical space (or at a vertex of the n-dimensional cube). This information is furnished by a *table of correspondences** (Table 1.6), which specifies our function in the form $y = y(k)$. A table of correspondences $y = y(k)$, together with a table for the n-dimensional binary logical space $k = k(x_1, x_2, \ldots, x_n)$ completely specifies the homogeneous binary logical function $y = y(x_1, x_2, \ldots, x_n)$ of n arguments.

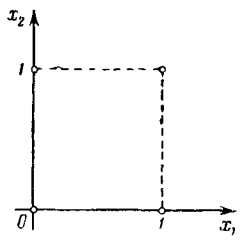


Fig. 1.3.

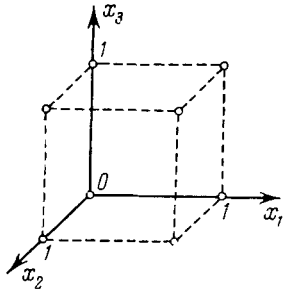


Fig. 1.4.

At any k, the function $y(k)$ can be either 0 or 1. It follows that each function of n arguments can be represented in a table of correspondences by some sequence of zeros and ones. The length of this sequence is $r = 2^n$, so that the total number of different functions that may possibly be constructed on the set of points of an n-dimensional binary space is $s = 2^r = 2^{(2^n)}$. All these functions can therefore be enumerated and hence designated by a numeral (numbered).

Table 1.6

k	0	1	2	3	...	$2^n - 2$	$2^n - 1$
y	$y(0)$	$y(1)$	$y(2)$	$y(3)$	...	$y(2^n - 2)$	$y(2^n - 1)$

There is a convenient method for scanning and numbering all these functions. This involves constructing a table such as 1.7 that combines all the possible correspondence tables; it contains $r = 2^n$ columns and $s = 2^r$ rows. We can complete this table as follows: We

*The term table of combinations is also used.

Table 1.7

y \ k	0	1	2	3	...	$r-1$
y_0	0	0	0	0	...	0
y_1	1	0	0	0	...	0
y_2	0	1	0	0	...	0
y_3	1	1	0	0	...	0
...	...	...	...	...	...	
y_{s-1}	1	1	1	1	...	1

enter pairs (01) into the first column, then we enter groups of four (0011) into the second column, groups of eight (00001111) into the third, and so on.* Then the string of zeros and ones in each row, when read from right to left, will be the binary representation of the number designating the function corresponding to this row. We shall refer to such a table as a *general correspondence table*.

In addition to this tabular method of defining homogeneous binary logical functions, there exists an analytical procedure which is widely used. This procedure is based on our ability to transform homogeneous functions into composite functions. Indeed, we know that both the homogeneous function and its argument assume values from the same set. This means that the logical variable that is the functional variable in one relation may be the argument in another relation. The transformation into composite functions allows us to express any homogeneous binary logical function in terms of certain simple functions. Naturally, the use of such functions and of the related notation entails some specific rules, that is, a special algebra.

b) Functions of One and Two Variables

Let us begin with the simplest case, where the function has only one argument ($n = 1$), and where the general correspondence table

*This table differs from 1.5 in that we fill in the columns and not the rows.

combined with the table of unidimensional binary logical space assumes the form of Table 1.8.

Table 1.8

k	0	1	Notation
x	0	1	
y_0	0	0	$y_0 = 0$
y_1	1	0	$y_1 = \bar{x}$
y_2	0	1	$y_2 = x$
y_3	1	1	$y_3 = 1$

The number of points in this logical space is then $r = 2^n = 2^1 = 2$, while the number of different functions is $s = 2^r = 2^2 = 4$. These four functions—y_0, y_1, y_2, and y_3— are shown in Table 1.8. There are no other functions of one argument.

The values of the functions y_0 and y_3 do not vary with the values of the argument, so that these functions are called *constant.* We shall denote them by $y_0 = 0$, $y_3 = 1$.

The function y_2, called the *identity function*, always assumes the same value as the argument x; the obvious notation is $y_2 = x$.

The function y_1 becomes 1 when $x = 0$, and 0 when $x = 1$. It is termed *negation*, and merits a special notation, $y_1 = \bar{x}$; this is read as "not x." Note that two of the above four functions can always be expressed as composite functions, using the symbolic notation for the two other functions. Thus,

$$\begin{aligned} y_3 &= \bar{y}_0 = \bar{0} = 1, \\ y_2 &= \bar{y}_1 = \bar{\bar{x}} = x. \end{aligned} \tag{1.1}$$

Therefore, we can define any homogeneous binary function of one argument in an analytical form by applying the special symbolic notation for negation to the two functions $y = 0$ and $y = \bar{x}$.

The general correspondence table for the case of functions of two arguments x_1 and x_2 $(n = 2)$, is Table 1.9. Here the number of points

Table 1.9

k	0	1	2	3	Notation
x_1	0	1	0	1	
x_2	0	0	1	1	
y_0	0	0	0	0	$y_0 = 0$
y_1	1	0	0	0	$y_1 = x_1 \downarrow x_2$
y_2	0	1	0	0	$y_2 = x_1 \leftarrow x_2$
y_3	1	1	0	0	$y_3 = \bar{x}_2$
y_4	0	0	1	0	$y_4 = x_2 \leftarrow x_1$
y_5	1	0	1	0	$y_5 = \bar{x}_1$
y_6	0	1	1	0	$y_6 = x_1 \nabla x_2$
y_7	1	1	1	0	$y_7 = x_1/x_2$
y_8	0	0	0	1	$y_8 = x_1 \& x_2$
y_9	1	0	0	1	$y_9 = x_1 \sim x_2$
y_{10}	0	1	0	1	$y_{10} = x_1$
y_{11}	1	1	0	1	$y_{11} = x_2 \rightarrow x_1$
y_{12}	0	0	1	1	$y_{12} = x_2$
y_{13}	1	0	1	1	$y_{13} = x_1 \rightarrow x_2$
y_{14}	0	1	1	1	$y_{14} = x_1 \vee x_2$
y_{15}	1	1	1	1	$y_{15} = 1$

in the logical space is $r = 2^n = 2^2 = 4$, while the number of different functions is $s = 2^r = 2^4 = 16$. The column on the extreme right gives the notation used for these functions. We see that six of these sixteen functions were encountered among the functions of one argument. These include two constant functions ($y_0 = 0$ and $y_{15} = 1$), two identity functions ($y_{10} = x_1$ and $y_{12} = x_2$), and two negation functions ($y_3 = \bar{x}_2$ and $y_5 = \bar{x}_1$).

Of the remaining ten functions, two (y_4 and y_{11}) are not independent, since they differ from functions y_2 and y_{13} only in the relative position of the two arguments. We are thus left with eight new functions of two independent variables. They have the following special properties:

The function $y_{14} = x_1 \vee x_2$ is 0 if, and only if, both arguments are 0. It is called *disjunction* and is read "x_1 or x_2."

The function $y_{13} = x_1 \rightarrow x_2$ is called *implication.* It becomes 0 if, and only if, the first argument (x_1) is 1 and the second (x_2) is 0; it is read "if x_1 then x_2" or "from x_1 follows x_2."

The function $y_9 = x_1 \sim x_2$ is called *equivalence.* It becomes 1 if both arguments have the same value, and it is 0 if the arguments have different values. It is read "x_1 is equivalent to x_2," or "x_1 if, and only if, x_2."

The function $y_8 = x_1 \,\&\, x_2$ becomes 1 if, and only if, both arguments are equal to 1. It is called *conjunction* and is read "x_1 and x_2."

The function $y_7 = x_1/x_2$ is called the *Sheffer stroke*; it is 0 if, and only if, both arguments are 1.

The function $y_6 = x_1 \bigtriangledown x_2$ is called the *Exclusive OR*; it is 1 if either the first or the second argument is 1 (but not if both are equal to 1).

The function $y_2 = x_1 \leftarrow x_2$ is called, in technical applications, the *inhibit function.* It is equal to the first argument (x_1) if the second argument (x_2) is 0; if the second argument is 1, the function becomes 0, no matter what the first argument is.

The function $y_1 = x_1 \downarrow x_2$ is called the *Pierce stroke function*; it becomes 0 if, and only if, both arguments are 0.

Now, we should also note that any function in the upper part of the table (that is, one of the functions $y_0, y_1, \ldots, y_7$) is a negation of some function from the lower part of the table (that is, one of the functions $y_8, y_9, \ldots, y_{15}$).

Consider, for example, the functions y_6 and y_9. We see from the table that $y_6 = 0$ if (and only if) $y_9 = 1$ and, conversely, $y_6 = 1$ if $y_9 = 0$. Thus, the variable y_6 may itself be considered an argument whose values uniquely determine the values assumed by variable y_9. From our definition of negation, we have $y_6 = \bar{y}_9$. But $y_6 = x_1 \nabla x_2$ and $y_9 = x_1 \sim x_2$. Consequently, $x_1 \nabla x_2 = \overline{x_1 \sim x_2}$. The table also shows that this relationship holds for all pairs of functions which are arranged symmetrically around a line dividing the seventh and eighth rows. We can write this relationship as $y_{15-i} = \bar{y}_i$, where $i = 0, 1, 2, \ldots, 15$.

Thus, the table implies that exactly half (i.e., four) of the eight two-argument functions still under discussion, are not independent.

Indeed,

$$\left.\begin{array}{llll} y_7 = \bar{y}_8, & \text{i.e.,} & x_1/x_2 = \overline{x_1 \,\&\, x_2}, \\ y_6 = \bar{y}_9, & \text{i.e.,} & x_1 \nabla x_2 = \overline{x_1 \sim x_2}, \\ y_2 = \bar{y}_{13}, & \text{i.e.,} & x_1 \leftarrow x_2 = \overline{x_1 \rightarrow x_2}, \\ y_1 = \bar{y}_{14}, & \text{i.e.,} & x_1 \downarrow x_2 = \overline{x_1 \vee x_2}. \end{array}\right\} \tag{1.2}$$

Therefore we can now drop operations defined by $/$, ∇, $\leftarrow$, and $\downarrow$, and obtain a list of six simple logical functions

$$\left.\begin{array}{ll} \text{constant} & y = 0, \\ \text{negation} & y = \bar{x}, \end{array}\right\}$$
$$\left.\begin{array}{ll} \text{conjunction} & y = x_1 \,\&\, x_2, \\ \text{disjunction} & y = x_1 \vee x_2, \\ \text{implication} & y = x_1 \rightarrow x_2, \\ \text{equivalence} & y = x_1 \sim x_2 \end{array}\right\} \tag{1.3}$$

which are sufficient, but by no means necessary, for expressing any function of one or two independent variables in an analytical form.

To prove this, consider the function $y = \bar{x}_1 \vee x_2$. Because it is a function of two independent variables, it must be equivalent to one of those shown in Table 1.9. We shall determine which one by finding its values at all four points of the corresponding two-dimensional binary logical space, that is, at all possible values of arguments x_1 and x_2. The process of finding these values is illustrated by Table 1.10, where we use the notation $y_1 = \bar{x}_1$; consequently $y = y_1 \vee x_2$. Our values of y show that $y = x_1 \rightarrow x_2$, which means that we are dealing with the identity

$$x_1 \rightarrow x_2 = \bar{x}_1 \vee x_2. \tag{1.4}$$

Similarly, it can be shown that

$$x_1 \rightarrow x_2 = \overline{x_1 \,\&\, \bar{x}_2}, \tag{1.5}$$

$$x_1 \sim x_2 = (\bar{x}_1 \vee x_2) \,\&\, (x_1 \vee \bar{x}_2). \tag{1.6}$$

Identities (1.5) and (1.6) show that functions of one or two independent variables may be completely described without employing implication and equivalence. Thus our set of simple functions may be further reduced to the following four:

$$\left.\begin{array}{ll} \text{constant} & y = 0, \\ \text{negation} & y = \bar{x}, \\ \text{conjunction} & y = x_1 \,\&\, x_2, \\ \text{disjunction} & y = x_1 \vee x_2. \end{array}\right\} \tag{1.7}$$

As we shall see below, this is the most convenient set of simple functions and hence is the one most frequently employed. However, in principle, even this set can be still further reduced.

Indeed, the procedure used to establish identities (1.4), (1.5), and (1.6), which enables us to dispense with implication and equivalence, may be employed to show that the following identities also hold:

$$\left.\begin{array}{l} x_1 \vee x_2 = \overline{\bar{x}_1 \,\&\, \bar{x}_2}, \\ x_1 \,\&\, x_2 = \overline{\bar{x}_1 \vee \bar{x}_2}, \\ 0 = x \,\&\, \bar{x}. \end{array}\right\} \tag{1.8}$$

Table 1.10

k	0	1	2	3
x_1	0	1	0	1
x_2	0	0	1	1
y_1	1	0	1	0
y	1	0	1	1

This means that either of the last two functions, as well as the first function of set (1.7) can also be dropped. We thus arrive at a set consisting of only two functions

$$\left.\begin{array}{ll} \text{negation} & y = \bar{x}, \\ \text{conjunction} & y = x_1 \,\&\, x_2 \\ \text{(or disjunction} & y = x_1 \vee x_2), \end{array}\right\} \tag{1.9}$$

by means of which we can express any function of one or two arguments.

We shall conclude this subsection by pointing out the special properties of the Sheffer stroke $y = x_1/x_2$ and the Pierce stroke $y = x_1 \downarrow x_2$. Either of these is sufficient for complete expression of any function of one or two independent variables by virtue of the fact that both functions of the previously described set (1.0) may be expressed by either of these forms. Thus

$$\left.\begin{array}{l} \bar{x} = x/x = x \downarrow x, \\ x_1 \,\&\, x_2 = (x_1/x_2)/(x_1/x_2), \\ x_1 \vee x_2 = (x_1 \downarrow x_2) \downarrow (x_1 \downarrow x_2). \end{array}\right\} \tag{1.10}$$

And since the set (1.9) is sufficient for complete description, so are the two special functions.

c) Functions of n Variables. Conjunctive and Disjunctive Normal Forms

The symbolism employed with one- and two-argument functions may be extended to functions of three, four, and, in general, n independent variables; for example,

$$y = (x_1 \to \bar{x}_2) \sim (\bar{x}_1 \,\&\, x_3). \tag{1.11}$$

We can construct a correspondence table for a function of n arguments. To complete it, we scan all possible combinations of the values $x_1, x_2, \ldots, x_n$ (that is, all the points $k_0, k_1, \ldots, k_{2^n-1}$ in an n-dimensional logical binary space) and determine the values of y at these points. For instance, for the function (1.11) at the point $k = 2$, that is, at $x_1 = 0$, $x_2 = 1$, $x_3 = 0$, we have $y(0, 1, 0) = 0$. Similar computations at all other points give Table 1.11.

Table 1.11

k	0	1	2	3	4	5	6	7
x_1	0	1	0	1	0	1	0	1
x_2	0	0	1	1	0	0	1	1
x_3	0	0	0	0	1	1	1	1
y	0	0	0	1	1	0	1	1

We shall now show that the symbolism used for functions of one or two arguments also allows us to express in analytical form any function of any number of independent variables.

As an example, consider Table 1.11. We shall assume that this table is given but that its analytical expression (1.11) is unknown, and we shall derive that expression from the table. In so doing we shall employ a procedure that is applicable to any other similar table.

Let us first consider any column in Table 1.11 in which $y = 1$: for instance, column $k = 3$. In this column $x_1 = 1$, $x_2 = 1$, $x_3 = 0$. We therefore write $y_1 = x_1 \,\&\, x_2 \,\&\, \bar{x}_3$, which, as can easily be seen, becomes 1 if, and only if, $x_1 = 1$, $x_2 = 1$, and $x_3 = 0$, that is, precisely at point $k = 3$. In an analogous manner we derive the functions

$$\left.\begin{aligned} y_2 &= \bar{x}_1 \,\&\, \bar{x}_2 \,\&\, x_3, \\ y_3 &= \bar{x}_1 \,\&\, x_2 \,\&\, x_3, \\ y_4 &= x_1 \,\&\, x_2 \,\&\, x_3, \end{aligned}\right\} \tag{1.12}$$

that become 1 only at points numbered $k = 4$, $k = 6$, and $k = 7$, respectively, that is, at those points of Table 1.11 at which $y = 1$.

Function $y = y_1 \vee y_2 \vee y_3 \vee y_4$ becomes 0 if, and only if, $y_1 = 0$, $y_2 = 0$, $y_3 = 0$ and $y_4 = 0$; in all other cases, $y = 1$. Since these "other cases" are the points $k = 3$, $k = 4$, $k = 6$, and $k = 7$, this means that function

$$y = (x_1 \& x_2 \& \bar{x}_3) \vee (\bar{x}_1 \& \bar{x}_2 \& x_3) \vee (\bar{x}_1 \& x_2 \& x_3) \vee (x_1 \& x_2 \& x_3) \quad (1.13)$$

corresponds exactly to our starting Table 1.11. We have thus obtained an analytical expression for the function given by Table 1.11. However, our new expression is not in the form of Eq. (1.11), but in another, "standard," form. While there is a marked difference in the appearance of (1.11) and (1.13), both expressions represent the same function, defined by Table 1.11; that is, we have the identity

$$\begin{aligned}(x_1 \to \bar{x}_2) \sim (\bar{x}_1 \& x_3) = (x_1 \& x_2 \& \bar{x}_3) \vee \\ \vee (\bar{x}_1 \& \bar{x}_2 \& x_3) \vee (\bar{x}_1 \& x_2 \& x_3) \vee (x_1 \& x_2 \& x_3).\end{aligned} \quad (1.14)$$

The technique just illustrated is quite general. Indeed, any function of n arguments can be given in the form of Table 1.12. Let us now take any column in which $y = 1$ and, writing out the conjunction of all the n independent variables $x_1 \& x_2 \& x_3 \& \ldots \& x_n$, let us mark with the sign of negation those variables of this column that become 0. We then form such conjunctions for all the other columns where $y = 1$, and we join them together by disjunction signs. Now we shall have an expression containing several conjunctive terms joined by disjunction signs. Each such term contains all the variables x_1, x_2, ..., x_n, some or all of which carry negation signs [for example, we may have $x_1 \& x_2 \& \ldots \& x_n$ (no negated variables), as well as $\bar{x}_1 \& \bar{x}_2 \& \ldots \& \bar{x}_n$ (all the variables negated)]. The various functions derived from the table and represented in this form can differ only in the number of disjunctive terms and in the way in which the negation signs are distributed above the variables x_i of the component conjunctions.

Expressions of this type are very important in propositional calculus: the disjunctive expression, constructed of terms which are different conjunctions of all the independent variables of a logical function, or their negations, is called the complete (or full, or perfect) *disjunctive normal form* of the function.

Table 1.12

k	0	1	2	3	...	k	...	$2^n - 1$
x_1	0	1	0	1	...	$x_1(k)$	...	1
x_2	0	0	1	1	...	$x_2(k)$	...	1
...	...	...	...	...	...	...	...	...
x_n	0	0	0	0	...	$x_n(k)$	...	1
y	$y(0)$	$y(1)$	$y(2)$	$y(3)$	...	$y(k)$	...	$y(2^n - 1)$

The complete (or full, or perfect) *conjunctive normal form* is the conjunctive expression constructed of terms which are different disjunctions of all the independent variables of a logical function, or their negations.

The term *complete* is usually omitted, that is, we speak of a *disjunctive normal* or a *conjunctive normal* form whenever it is not required that *each* term of such a form be a conjunction or a disjunction (as the case may be) of all the variables of a logical function.

Let us now consider the following property of normal forms. If a function y is expressed by a normal (either simple or full) disjunctive (or conjunctive) form, and if all the $\vee$ signs in this expression are replaced by & and all the & signs are replaced by $\vee$, and if a negation sign is placed above each variable (if the variable already carries such a sign, another identical sign is added to it; this is equivalent to a removal of negation), then we obtain function $\bar{y}$ written in normal (either simple or full) conjunctive (or disjunctive) form. This property is a direct consequence of identities (1.8).*

In contrast to simple normal forms, the full normal forms are unique in the sense that there is only one way in which each function can be represented as a full normal disjunctive or conjunctive form (that is, if we disregard permutations of disjunctive or conjunctive terms and of independent variables).

We shall illustrate the importance of these concepts by two problems.

*This is referred to as Duality or De Morgan's Law.

Problem 1. Determine whether a function n arguments $y = y(x_1, x_2, \ldots, x_n)$ can be reduced to a constant function $y = 0$.

This problem is solved by reducing the given function to its disjunctive normal form. Then, if one finds that each disjunctive term contains at least one variable in conjunction with its negation (that is, $x_i \& \bar{x}_i$), the function is of the form $y = 0$. If this is not the case, than we can always find values at which $y = 1$; that is, the function is not a constant $y = 0$.

This problem has a dual in which it is required to determine whether a given function can be reduced to the form $y = \bar{0} = 1$. The solution is obtained by reducing the given function to its conjunctive normal form. If one then finds that each conjunctive term contains the expression $x_i \vee \bar{x}_i$, then in this case (and only in this case) the given function reduces to the form $y = 1$.

The question whether some function $y = y(x_1, x_2, \ldots, x_n)$ can be reduced to the form $y = 1$ or $y = 0$ is called the *decision problem.* Within this problem, functions that reduce to the form $y = 1$ (or $y = 0$) are called *identically true* (or *false*), whereas functions that do not reduce to either $y = 1$ or $y = 0$ are called *feasible.*

Problem 2. Given a logical function of n arguments $y = y(x_1, x_2, \ldots, x_n)$, find all sets of values of arguments at which $y = 1$.

The problem would be solved if the given function could be reduced to its full disjunctive normal form.

The required number of sets of argument values is exactly equal to the number of disjunctive terms in the full disjunctive normal form of the function. The specific values of all the arguments in each set is determined in the following manner. Each set of values x_i (where $i = 1, 2, \ldots, n$) at which $y = 1$ (the values are defined by the jth parentheses) has the form

$$x_1 = x_{1j}, \quad x_2 = x_{2j}, \quad \ldots, \quad x_n = x_{nj},$$

where x_{ij} is equal to 0 or 1, depending on whether the corresponding ith independent variable appears in the jth conjunctive parenthesis with or without a negation sign.

d) Functions of n Variables. The Algebra of Propositional Calculus

The full disjunctive normal form (1.13) of our example defined a function for which we already had a shorter expression. In other cases, too, there exist functional expressions that are shorter and more convenient to use than the full disjunctive normal forms. In

other words, there are other cases in which we can establish identities similar to (1.14).

Thus far, we have proved all identities of functions of one, two, or more variables by a test involving substitution of all the possible values of these variables. This method has two major disadvantages: it does not afford any opportunities for deriving new identities and in this sense is passive; in addition, it becomes more and more laborious as the number of variables increases. Fortunately, however, we have at our disposal another method based on the use of certain rules for identical transformations. Thus the collection of simple functions

$$y = 0, \quad y = \bar{x}, \quad y = x_1 \& x_2, \quad y = x_1 \vee x_2, \quad y = x_1 \to x_2, \quad y = x_1 \sim x_2 \tag{1.15}$$

may be operated upon by means of a system of rules, usually referred to as the *algebra of logic* or *Boolean algebra*, which consists of the following identities:

$$\bar{\bar{x}} = x, \tag{1.16}$$

$$x_1 \to x_2 = \bar{x}_1 \vee x_2, \tag{1.17}$$

$$x_1 \sim x_2 = (x_1 \to x_2) \& (x_2 \to x_1), \tag{1.18}$$

$$\text{(a) } x \& x = x, \qquad \text{(b) } x \vee x = x, \tag{1.19}$$

$$\text{(a) } x \& \bar{x} = 0 \qquad \text{(b) } x \vee \bar{x} = 1, \tag{1.20}$$

$$\text{(a) } x \& 1 = x, \qquad \text{(b) } x \vee 1 = 1, \tag{1.21}$$

$$\text{(a) } x \& 0 = 0, \qquad \text{(b) } x \vee 0 = x, \tag{1.22}$$

$$\text{(a) } \overline{x_1 \& x_2} = \bar{x}_1 \vee \bar{x}_2, \qquad \text{(b) } \overline{x_1 \vee x_2} = \bar{x}_1 \& \bar{x}_2, \tag{1.23}$$

$$\text{(a) } x_1 \& x_2 = x_2 \& x_1, \qquad \text{(b) } x_1 \vee x_2 = x_2 \vee x_1, \tag{1.24}$$

$$\text{(a) } x_1 \& (x_2 \& x_3) = (x_1 \& x_2) \& x_3, \qquad \text{(b) } x_1 \vee (x_2 \vee x_3) = (x_1 \vee x_2) \vee x_3, \tag{1.25}$$

$$\text{(a) } x_1 \& (x_2 \vee x_3) = (x_1 \& x_2) \vee (x_1 \& x_3), \qquad \text{(b) } x_1 \vee (x_2 \& x_3) = (x_1 \vee x_2) \& (x_1 \vee x_3). \tag{1.26}$$

Each of these identities may be proved by direct substitution of all the possible values of the variables appearing in the left and the right sides of the identity.

The OR and AND operations of this algebra have much in common with addition and multiplication of ordinary algebra. Thus, they obey the first and second commutative laws [identities (1.24)], as well as the first and second associative laws [identities (1.25)]. However, in contrast to ordinary algebra, they obey two distributive laws rather than one [identities (1.26)]; and "reduction of like terms" or "multiplication of a variable by itself" are accomplished via identities (1.19), without introducing any factors or exponents.

This system of identities permits a purely analytical solution of a great variety of problems. Moreover, standard methods may be used for some of these solutions. For instance, any analytical function may be transformed directly into a normal disjunctive form, a procedure illustrated by the following example.

EXAMPLE. Let the starting function be

$$y = \overline{|x_1 \to (\bar{x}_1 \sim x_3) \,\&\, (x_2 \to \bar{x}_3)|} \vee \overline{(x_1 \to x_3)}. \tag{1.27}$$

First, let us eliminate the $\to$ and $\sim$ signs. Applying identities (1.17) and (1.18), we obtain

$$y = \overline{|\bar{\bar{x}}_1 \vee \overline{[(\bar{\bar{x}}_1 \vee x_3) \,\&\, \overline{(\bar{x}_3 \vee \bar{x}_1)}]} \,\&\, (\bar{x}_2 \vee \bar{x}_3)|} \vee \overline{(\bar{\bar{x}}_1 \vee x_3)}. \tag{1.28}$$

Next, let us eliminate those negation signs that relate not just to a single variable but to an entire aggregation of such variables. We do this by means of identities (1.23) and (1.16) to obtain

$$\begin{aligned} y &= [\bar{\bar{x}}_1 \,\&\, \overline{(\bar{\bar{x}}_1 \vee x_3) \,\&\, \overline{(\bar{x}_3 \vee \bar{x}_1)}} \,\&\, (\bar{x}_2 \vee \bar{x}_3)] \vee (\bar{\bar{x}}_1 \,\&\, \bar{x}_3) = \\ &= |x_1 \,\&\, \overline{(\bar{x}_1 \vee x_3)} \vee \overline{(\bar{\bar{x}}_3 \vee \bar{x}_1)} \,\&\, (\bar{x}_2 \vee \bar{x}_3)] \vee (x_1 \,\&\, \bar{x}_3) = \\ &= [x_1 \,\&\, [(\bar{x}_1 \,\&\, \bar{x}_3) \vee (x_3 \,\&\, x_1)] \,\&\, (\bar{x}_2 \vee \bar{x}_3)\} \vee (x_1 \,\&\, \bar{x}_3). \end{aligned} \tag{1.29}$$

Now, in order to arrive at a disjunctive normal form it is sufficient to expand the expression in the braces as specified by identity (1.26a). Simplifying the intermediate results by means of (1.16) and (1.19) as we go along, we obtain

$$\begin{aligned} y &= \{|(x_1 \,\&\, \bar{x}_1 \,\&\, \bar{x}_3) \vee (x_1 \,\&\, x_3 \,\&\, x_1)] \,\&\, (\bar{x}_2 \vee \bar{x}_3)\} \vee (x_1 \,\&\, \bar{x}_3) = \\ &= (x_1 \,\&\, \bar{x}_1 \,\&\, \bar{x}_3 \,\&\, \bar{x}_2) \vee (x_1 \,\&\, x_3 \,\&\, \bar{x}_2) \vee (x_1 \,\&\, \bar{x}_1 \,\&\, \bar{x}_3 \,\&\, \bar{x}_3) \vee \\ &\qquad \vee (x_1 \,\&\, x_3 \,\&\, \bar{x}_3) \vee (x_1 \,\&\, \bar{x}_3). \end{aligned} \tag{1.30}$$

The first, third, and fourth disjunctive terms of the above disjunctive normal form are 0, since they contain expressions of the form

$x \& \bar{x}$. The second and last terms lack such expressions, and therefore our function does not reduce to $y = 0$; it may therefore be written as

$$y = (x_1 \& \bar{x}_2 \& x_3) \vee (x_1 \& \bar{x}_3). \tag{1.31}$$

Thus,

$$\begin{aligned} y &= \overline{[x_1 \to (\bar{x}_1 \sim x_3) \& (x_2 \to \bar{x}_3)]} \vee \overline{(\bar{x}_1 \to x_3)} = \\ &= (x_1 \& \bar{x}_2 \& x_3) \vee (x_1 \& \bar{x}_3). \end{aligned} \tag{1.32}$$

To reduce a given function to its complete disjunctive normal form, it must first be reduced to some normal disjunctive form by the methods already discussed. Let us follow the remainder of the procedure on our example.

Our normal form (1.32) is not full because its second disjunctive term does not comprise all the variables: x_2 (or $\bar{x}_2$) is missing. However, it is readily seen that the following identity is true:

$$x_1 \& \bar{x}_3 = x_1 \& \bar{x}_3 \& (x_2 \vee \bar{x}_2) = (x_1 \& x_2 \& \bar{x}_3) \vee (x_1 \& \bar{x}_2 \& \bar{x}_3). \tag{1.33}$$

Substituting the disjunction given by (1.33) for the second disjunctive term of the normal form (1.32), we obtain

$$y = (x_1 \& \bar{x}_2 \& x_3) \vee (x_1 \& x_2 \& \bar{x}_3) \vee (x_1 \& \bar{x}_2 \& \bar{x}_3) \tag{1.34}$$

which is the full disjunctive normal form of the given function. Of course, if these transformations would have given an expression containing several identical disjunctive terms, we would have retained only one of these.

Reduction to a conjunctive normal form differs from the above technique only in the last step where, instead of expanding the expression derived in the preceding steps in accordance with identity (1.26a), we use the second distributive law, that is, identity (1.26b).

EXAMPLE. Let the conjunctive normal form of a function of three variables be

$$y = x_1 \& (\bar{x}_1 \vee \bar{x}_2 \vee x_3). \tag{1.35}$$

To transform this into a complete normal form, we use the identities

$$\left.\begin{aligned} x_1 &= x_1 \vee (x_2 \,\&\, \bar{x}_2) = (x_1 \vee x_2) \,\&\, (x_1 \vee \bar{x}_2), \\ x_1 \vee x_2 &= x_1 \vee x_2 \vee (x_3 \,\&\, \bar{x}_3) = (x_1 \vee x_2 \vee x_3) \,\&\, (x_1 \vee x_2 \vee \bar{x}_3), \\ x_1 \vee \bar{x}_2 &= x_1 \vee \bar{x}_2 \vee (x_3 \,\&\, \bar{x}_3) = (x_1 \vee \bar{x}_2 \vee x_3) \,\&\, (x_1 \vee \bar{x}_2 \vee \bar{x}_3). \end{aligned}\right\} \quad (1.36)$$

and we get

$$\begin{aligned} y = (x_1 \vee x_2 \vee x_3) \,\&\, (x_1 \vee x_2 \vee \bar{x}_3) \,\&\, (x_1 \vee \bar{x}_2 \vee x_3) \,\& \\ \&\, (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \,\&\, (\bar{x}_1 \vee \bar{x}_2 \vee x_3), \end{aligned} \quad (1.37)$$

which is the complete conjunctive normal form of the starting function.

So far, we have demonstrated the use of Boolean algebra by reducing a given logical function of several variables to its full normal (or any normal) disjunctive (or conjunctive) form. However, the function may be given by means of a table. In this case, we can obtain a unique complete disjunctive normal form via the method already described. We can then transform this expression by means of Boolean identities and thus arrive at other analytic expressions of the same function. Given the variety of possible forms of a function, we are faced with the problem of determining which of these is optimum for our purposes. We shall return to this somewhat later.

1.4. TWO-VALUED PREDICATE CALCULUS

We shall now return to a subject which we have briefly considered at the end of Section 1.2. Thus we shall consider two-valued predicates, that is, logical functions which themselves assume only the values of 0 or 1, but whose arguments may take on values from any set whatsoever.

Predicate functions are denoted by capital letters. This permits us to distinguish visually between a predicate (a nonhomogeneous logical function) and a complex proposition (a homogeneous logical function). Thus, an n-place predicate may be written as

$$y = P(x_1, \ldots, x_n),$$

where $x_1 = \{x_{11}, \ldots, x_{1p}\}, \ldots, x_n = \{x_{n1}, \ldots, x_{nq}\}$ are the object variables and their alphabets.

Since two-valued predicates assume values from a binary set 0 and 1, they may themselves be the arguments of two-valued

homogeneous logical functions; for this reason, we can apply to them the symbolism of propositional calculus. Thus, suppose we have the predicates

$$\left.\begin{aligned} y_1 &= P(x_1), \text{ where } x_1 = \{x_{11}, \ldots, x_{1p}\}, \\ y_2 &= Q(x_2), \text{ where } x_2 = \{x_{21}, \ldots, x_{2q}\}. \end{aligned}\right\} \tag{1.38}$$

We can subject these predicates to any one of the operations of propositional calculus to obtain a new predicate; for example,

$$R(x_1, x_2) = P(x_1) \vee Q(x_2). \tag{1.39}$$

This use of operations of propositional calculus permits us to achieve several ends. To begin with, we can relate several simple predicates to each other and form a compound predicate, as in the above example. Also, we can relate predicates to any and all simple propositions, as well as to the compound propositions that can be formed from the simple ones by the same operations of propositional calculus. Thus from the predicates (1.38) and the binary logical variables

$$x_3 = \{0, 1\}, \quad x_4 = \{0, 1\}$$

we can construct a composite function, for example,

$$z = \{P(x_1) \rightarrow [Q(x_2) \vee (x_3 \& \bar{x}_4)]\} \sim x_3 \tag{1.40}$$

where Z can only be two-valued.

The only variables which we have encountered in the compound function of propositional calculus were the simple prepositions. In the predicate calculus, however, not only simple propositions, but also the object variables of the predicates, as well as *variable predicates* can act as variables. The presence of these elements constitutes the main characteristic of this calculus, and necessitates new operations that are qualitatively different from those employed in propositional calculus. The operators corresponding to these new operations are called *quantifiers.*

There are two types of quantifiers: the *universal* and the *existential.*

The universal quantifier is an operator that matches any one-place predicate $y = P(x)$ with the binary logical variable Z which becomes 1 if, and only if, $y = 1$ at all values of x. This is written

$$z = (\forall x) P(x),$$

where "$\forall x$" is the universal quantifier. The above expression is then read as "for all x there is $P(x)$."

The existential quantifier is an operator that matches a one-place predicate $y = P(x)$ with a binary logical variable z which becomes 0 if, and only if, $y = 0$ at all values of x. This is written

$$z = (\exists x) P(x),$$

where "$\exists x$" is the existential quantifier. The above expression is then read as "there is an x such that $y = P(x)$."

Let us discuss some general properties of these operators. In accordance with the definitions of quantifiers, the logical variable z in

$$\left.\begin{aligned} z &= (\forall x) P(x), \\ z &= (\exists x) P(x) \end{aligned}\right\} \tag{1.41}$$

is not a function of the object variable x; here, z is an "integral" characteristic of the predicate $P(x)$. To underscore the absence of functional dependence of z on x, the object variable x in such cases is said to be *bound*. Object variables that are not bound are said to be *free*. Of course, the universal and existential quantifiers may also be applied to functions of propositional calculus. But if we do that, then they degenerate into finite conjunctions and disjunctions. Indeed, suppose we have a function $y = y(x_1, \ldots, x_n)$ in which both the variables and the function are two-valued logical variables. The same function may be given in the form $y = y(k)$, where k is a numeral denoting a point in an n-dimensional binary logical space. From the definition of quantifiers, we have

$$(\forall k)\, y(k) = y(0) \& y(1) \& \ldots \& y(k) \& \ldots \& y(2^n - 1),$$
$$(\exists k)\, y(k) = y(0) \vee y(1) \vee \ldots \vee y(k) \vee \ldots \vee y(2^n - 1).$$

For this reason we can consider the universal and existential quantifiers as generalized conjunction and generalized disjunction, respectively. And because of the analogy between conjunction or disjunction and the summation of real numbers, one can draw an analogy between the operations specified by quantifiers and the integration of functions of a real variable. If one applies a quantifier (either universal or existential) to an m-place (rather than a one-place) predicate, the result is again a predicate; this time it is, however, an $(m - 1)$-place predicate since one object variable becomes bound.

Thus, in dealing with predicates, we employ not only the operations of propositional calculus, but also operations involving binding of object variables by universal and existential quantifiers. The calculus in which the above operations are used to construct compound functions is called *restricted predicate calculus.*

This new operation of binding by quantifiers introduces identities which differ from those of the Section 1.3. Examples of such identities are

$$\overline{(\forall x)\,P(x)} = (\exists x)\,\overline{P(x)}, \tag{1.42a}$$

$$\overline{(\exists x)\,P(x)} = (\forall x)\,\overline{P(x)}. \tag{1.42b}$$

The identities of propositional calculus, supplemented by identities (1.42), comprise a mechanism useful for solving a variety of problems. As in propositional calculus, the most important problem of predicate calculus is that of decision, but because the independent variables are different, the manner in which this problem posed is also somewhat different.

Thus, the decision problem of propositional calculus in determining whether a given compound function is identically true, feasible, or identically false. However, the following must be asked in predicate calculus: (a) Is a given compound function identically true; that is, does it assume the value of 1 with any object variable and any predicate? Or (b) Is it identically true only over a certain set of object variables; that is, does it assume the value of 1 only over a certain set of object variables and for any predicate from this set? Or (c) Is it feasible; that is, does it assume the value of 1 at some values of object variables and at some predicates? And, finally, (d) Is it identically false, that is, unfeasible? In contrast to the case of propositional calculus, the decision problem of predicate calculus can be solved only for special kinds of compound functions.

2

Engineering Applications of Propositional Calculus

2.1. COMBINATIONAL RELAY SWITCHING CIRCUITS

We have already said that the promise of mathematical logic in engineering design first became apparent during the analysis of electrical relay switching circuits. In time, it became progressively more evident that this logic is not only applicable to *the analysis of relay switching* circuits but that the operation of such circuits mirrors the postulates of the logic. The result of this discovery was the relay switching theory. Then, when contactless devices that perform the same functions as relay switches came into existence, the special *theory of relay circuits* was extended into a general theory of switching systems.

We shall now examine this most conspicuous example of application of logic to engineering, concentrating on the so-called combinational relay switching circuits.

Every electric relay switching circuit contains two types of converters: electrical-to-mechanical and *mechanical-to-electrical*. The electromechanical relay converts electrical input signals into a mechanical displacement of its contacts. On the other hand, the mechanical-to-electrical converter is an electrical network comprising contacts and relay coils: it converts the mechanical displacement of its (input) contacts into electrical output signals (currents flowing in the coils of the relays). Connection of outputs of converters of one type to the inputs of converters of the other type gives a variety of relay switching networks.

The simplest electromechanical relay consists of a coil *1*, a core *2*, an armature *3*, and two groups of contacts: normally closed *4'*, and normally open *4''* (Fig. 2.1,a). If a current larger than the actuating current i_2 (Fig. 2.1, b) flows in the coil, the armature is

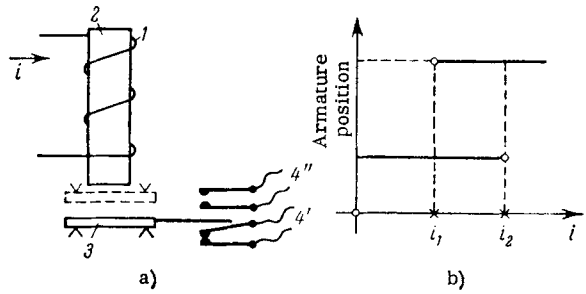


Fig. 2.1.

attracted to the core, and this causes all the normally open contacts to close and all the normally closed contacts to open. On the other hand, if a current smaller than i_1 flows in the coil (in particular, when the coil is de-energized, the armature recedes from the core, causing the normally closed contacts to close and the normally open contacts to open. To avoid the complications arising in transient states, we shall consider only the stable states of the relay, that is, states when the coil current i has either of the following values: $i < i_1$ or $i > i_2$. Thus the relay has two stable states or, to say it in another way, there are two states which are characteristic for all the elements of the relay. We are, however, interested only in the coil and in the contacts. Each contact has two states: closed and open. We shall designate these states by 1 and 0, respectively; that is, we shall treat the state of a contact as a binary logical variable assuming these values.

The coil also has two states. The first of these occurs at $i > i_2$, and we denote it by 1. In the second state, denoted by 0, the coil is de-energized ($i < i_1$). Thus the coil states can also be represented by a logical variable that can be either 0 or 1. The physical significance of these values is shown in Table 2.1.

Table 2.1

Component	Symbol	
	1	0
Contact	closed	open
Coil	energized	de-energized

The state of a relay contact is goverened by the state of the coil in the following manner:

For a normally open contact $x = X$,
For a normally closed contact $x' = \overline{X} = \bar{x}$.

where X, x, x' are the logical variables specifying the states of the coil, of the normally open contact and of the normally closed contact, respectively.

If a relay has more than one coil, it becomes convenient to an "equivalent coil." Thus, let the relay have two coils (X_1 and X_2) and let these be so connected that the relay shall operate only if both are energized, that is, if $X_1 \& X_2 = 1$. Letus now imagine a coil X_e, such that it will cause the relay to operate only if $X_1 \& X_2 = 1$. Obviously the action of our equivalent coil, which is related to that of the actual coils by

$$X_e = X_1 \& X_2$$

and which governs the state of the relay contacts in accordance with

$x = X_e$ (for normally open contacts)
$x' = \overline{X}_e = \bar{x}$ (for normally closed contacts),

is completely equivalent to the action of the two actual coils.

So far, we have discussed a relay with two coils. By the same reasoning, we can imagine a relay with m coils connected so that the relay operates only at certain combinations of the 1 and 0 states of the constituent coils.

A relay circuit incorporating m coils in a specific arrangement is described by a logical function specific to this circuit:

$$X_e = M(X_1, X_2, \ldots, X_m).$$

However, this specificity does not change the general relationship between the contacts of the relay and its equivalent coil.

We shall now turn to the representation of the mechanical-to-electrical converter, that is, of contacts connected to relay coils. Let us start with the case when the circuit (Figs. 2.2, a and 2.2,a′) consists of a contact *1* of an input relay and a coil *2* of another relay (the output relay), the coil being either in series (Fig. 2.2,a) or in parallel (Fig. 2.2,a′) with the contact.

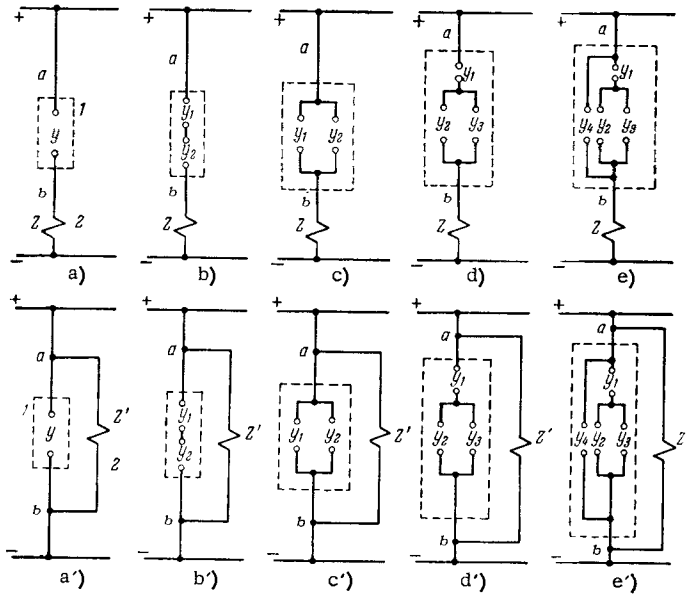


Fig. 2.2.

Retaining the symbols 0 and 1 for the states of the contacts and the coils, we obtain $Z = y$ for a coil connected in series and $Z' = \overline{y} = \overline{Z}$ for a coil connected in parallel; here, y is a logical variable specifying the state of the contact, and Z and Z' specify the states of the coils connected with it in series and in parallel, respectively.

In the usual practical case, we do not deal with a single contact I but with a group of contacts $y_1, y_2, \ldots, y_n$ belonging to several relays, all combined into an electrical network. We shall call an electrical network incorporating contacts a *switching network.* The dashed lines in Figs. 2.2,b-e enclose examples of such networks.

Again, it will be convenient to use an equivalent variable—the equivalent contact y_e—whose properties are analogous to those of the equivalent coil in a multiple-coil relay. For example, with the two contacts y_1 and y_2 of Fig. 2.2,b connected in series, the circuit in section ab will be closed only if $y_1 \& y_2 = 1$. By introducing the equivalent contact

$$y_e = y_1 \& y_2$$

and setting up the relationships

$$Z = y_e \qquad \text{for series connection Fig. 2.2,b)}$$

$$Z' = \bar{y}_e = \bar{Z} \text{ for parallel connection (Fig. 2.2,b')},$$

we preserve all the characteristics of the circuit.

Figure 2.2 also shows some other ways of arranging contacts into circuits; we thus obtain functions

$$y_e = y_1 \vee y_2 \qquad \text{(Fig. 2.2,c,c')},$$
$$y_e = y_1 \,\&\, (y_2 \vee y_3) \qquad \text{(Fig. 2.2,d,d')},$$
$$y_e = [y_1 \,\&\, (y_2 \vee y_3)] \vee y_4 \qquad \text{(Fig. 2.2,e,e')}.$$

In the general case,

$$y_e = N(y_1, y_2, \ldots, y_n);$$

In this generalized function, the relationship between the coils and the equivalent contact remains the same as that specified above (the matching condition N must, of course, reflect the actual arrangement of the contacts in the switching network when the function is used to represent a specific circuit). The physical meaning of y_e is that of the conductivity of a two-terminal network containing the given switching circuit.

Each of our converters has the ability to detect; that is, it exhibits a directional effect. In the electromechanical converter—the "relay with contacts"—the contact state is governed by the coil state, but the contacts have no effect on the coil state. In the mechanical-to-electrical converter—the "contact network with coils"—the coil state is governed by the states of the contacts, on which the coil has no effect. This property of converters allows us to treat them as devices with variable inputs and outputs. The input variables $X_1, \ldots, X_m$ of the electromechanical converter are the states of the relay coils (energized or de-energized); the two output variables x and x' of this device are the states of the two different contacts (normally open or closed). As already stated, this device may be treated as consisting of two series-connected subunits: The first performs the logical function

$$X_e = M(X_1, \ldots, X_m),$$

while the second realizes the functions

$$x = X_e, \quad x' = \bar{X}_e = \bar{x}.$$

In the mechanical-to-electrical converter the states of the contacts of the input relay act as input variables $y_1, \ldots, y_n$, and the coil states of the two output relays act as output variables Z and Z'. This device may again be treated as consisting of two series-connected subunits, the first of which realizes the logical function

$$y_e = N(y_1, \ldots, y_n),$$

while the second performs the functions

$$Z = y_e, \quad Z' = \bar{y}_e = \bar{Z}.$$

We see now that the two types of converters have identical properties. Any relay switching circuit may be broken down into units having the above-described properties.

We shall assume that our combinational relay switching circuits obey the following conditions: they consist of instantaneously actuated, ideal relays; and they have no feedback loops; that is, they consist only of subunits exhibiting a direction effect.

Figure 2.3,a shows the schematic of such a combinational relay switching circuit and Fig. 2.3,b shows the corresponding block diagram; it can be seen that the latter has no feedback loops. Contrast this with the schematic diagram shown in Fig. 2.4,a: its block diagram (Fig. 2.4,b) does show a feedback loop. The operation of such a circuit cannot be analyzed without taking into account the relay-actuating time.

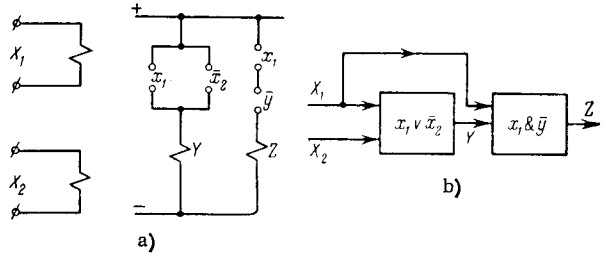


Fig. 2.3.

In drawing relay switching circuits we usually identify each coil by means of the corresponding logical variable; contacts are usually identified by an expression that specifies only the contact state (in terms of the state of the coil governing it).

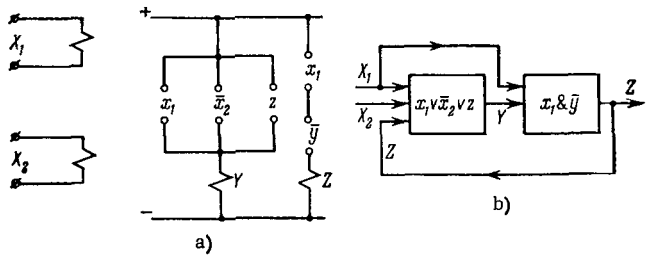


Fig. 2.4.

2.2. ANALYSIS OF COMBINATIONAL RELAY SWITCHING CIRCUITS

Consider the following problem: Given a combinational relay switching circuit, it is required to find its mathematical description, that is, to determine the logical function performed by this circuit.

We shall analyze only circuits with single-coil relays (these are the most common switching circuits). We have already seen (Section 2.1) that for these relays

$$x = X \text{ for in the case of normally open contacts}$$

$$x' = \bar{X} = \bar{x} \text{ for in the case of normally closed contacts,}$$

where X, x, and x' are variables specifying, respectively, the states of the coil, the normally open contact, and the normally closed contact.

This means that a single-coil relay operating alone will perform either repetition or negation.

The great variety of combinational switching circuits that can be synthesized is due to use of different arrangements of normally open and normally closed contacts of single-coil relays. The resulting switching network can be represented by the relationship

$$x_e = F(x_1, x'_1, \ldots, x_n, x'_n),$$

where x_i, x'_i, and x_e are variables specifying, respectively, the states of the contacts of the ith relay and of the equivalent contact.

However complex and involved a switching network may be, it is always possible to represent it in terms of

$$x_e = F(x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n).$$

Such a formula may be derived by a procedure which generalizes the obvious fact that if two contacts x_1 and x_2 are connected in series, we have $x_e = x_1 \& x_2$, while for contacts connected in parallel we have $x_e = x_1 \vee x_2$.

To clarify the principles on which this procedure is based, consider an example. The two-terminal switching circuit to be analyzed is shown in Fig. 2.5,a. We shall gradually simplify this circuit by introducing equivalent contacts. To start with, we shall eliminate all the chains of series-connected contacts. Putting

$$x_6 = x_3 \& \bar{x}_5; \quad x_7 = x_3 \& x_4; \quad x_8 = x_5 \& x_1;$$
$$x_9 = x_2 \& \bar{x}_1; \quad x_{10} = x_2 \& x_5; \quad x_{11} = x_3 \& \bar{x}_2; \quad x_{12} = x_2 \& x_5,$$

we transform the original circuit into its equivalent shown in Fig. 2.5,b.

The next step is to eliminate all the groups of contacts connected in parallel. To do this we write

$$x_{13} = x_2 \vee x_7; \quad x_{14} = \bar{x}_1 \vee \bar{x}_3; \quad x_{15} = x_8 \vee x_6; \quad x_{16} = x_9 \vee x_1,$$

or, using the notation already introduced,

$$x_{13} = x_2 \vee (x_3 \& x_4); \qquad x_{14} = \bar{x}_1 \vee \bar{x}_3;$$
$$x_{15} = (x_1 \& x_5) \vee (x_3 \& \bar{x}_5); \quad x_{16} = x_1 \vee (x_2 \& \bar{x}_4).$$

We then obtain the circuit shown in Fig. 2.5,c.

Again, we shall eliminate the chains of series-connected contacts in this new circuit. We do this by means of the following equivalents:

$$x_{17} = x_1 \& x_{13} = x_1 \& [x_2 \vee (x_3 \& x_4)] = (x_1 \& x_2) \vee (x_1 \& x_3 \& x_4),$$
$$x_{18} = x_{14} \& x_{15} = (\bar{x}_1 \vee \bar{x}_3) \& [(x_3 \& \bar{x}_5) \vee (x_1 \& x_5)] =$$
$$= (\bar{x}_1 \& x_3 \& \bar{x}_5) \vee (x_1 \& \bar{x}_3 \& \bar{x}_5),$$
$$x_{19} = x_{16} \& x_{10} = [x_1 \vee (x_2 \& \bar{x}_4)] \& x_2 \& x_5 =$$
$$= (x_1 \& x_2 \& x_5) \vee (\bar{x}_4 \& x_2 \& x_5).$$

We thus obtain the circuit shown in Fig. 2.5,d. This circuit cannot be further simplified by the above methods of elimination.

Now, let us number all the modes of this circuit, using identical numbers for those nodes which are directly interconnected (without intervening contacts). We thus have Fig. 2.5,d, with nodes 1, 2, ..., m (in our case, $m = 4$). It is now convenient to transform this circuit

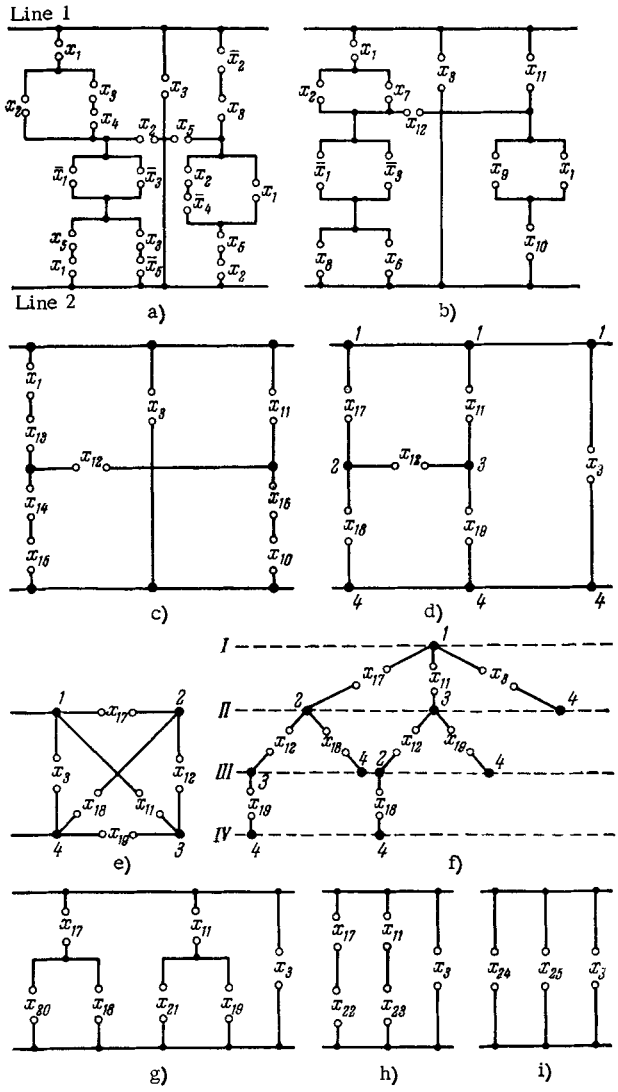


Fig. 2.5.

into the form of Fig. 2.5,e, which is obtained by combining all the nodes bearing identical numbers.

The circuit in Fig. 2.4,e can also be presented as a "tree" (Fig. 2.5,f) constructed in the following manner. We draw several tiers and assign to them the numbers corresponding to the m nodes of Fig. 2.5,e; we thus have tiers *I*, *II*, *III*, and *IV*. Node *1* is placed in

the first tier. From it we draw a cluster of $m-1$ branches, all terminating in the second tier. The ends of these branches are marked with the numbers of the remaining nodes of the circuit [that is, nodes which together with the initial node of the cluster (node 1) constitute the complete set]; in our case, that means the numbers 2, 3, and 4.

Next, we use each of these $m-1$ nodes of the second tier (but not the node m, that is, 4) as the origin of a cluster of $m-2$ branches which terminate in the third tier. Thus, we obtain two clusters of branches, originating at nodes 2 and 3, respectively. The third-tier ends of the cluster drawn from node 2 are given the numbers of all the second-tier nodes, with the exception of 2 (that is, the ends of this cluster carry the numbers 3 and 4). In the same way, we number the terminals of the cluster of $m-2$ branches originating at node 3 (but here we omit 3), and so on.

In the next step, each third-tier node, except those designated by m, serves as the origin of a cluster of $m-3$ branches joining the third and the fourth tiers. The fourth-tier terminals are numbered by the same procedure as the third. It is readily seen that the last (or mth) tier will now hold only nodes designated by m, from which no further branches can be drawn (dead-end nodes).

We now have a "tree" in which each "branch" connecting two nodes corresponds to the wire performing the same function in the circuit of Fig. 2.5,e.

The switching network of Fig. 2.5,f traces all the paths leading from node 1 to node 4 and is equivalent to that of Fig. 2.5,e. Now, we can eliminate all groups of series-connected contacts by using

$$x_{20} = x_{12} \& x_{19} = (x_1 \& x_2 \& x_5) \vee (\bar{x}_4 \& x_2 \& x_5),$$
$$x_{21} = x_{12} \& x_{18} = x_1 \& x_2 \& x_5 \& \bar{x}_3$$

and we get the circuit shown in Fig. 2.5,g. We then eliminate the groups of parallel contacts by using

$$x_{22} = x_{20} \vee x_{18} =$$
$$= (x_1 \& x_2 \& x_5) \vee (\bar{x}_4 \& x_2 \& x_5) \vee (\bar{x}_1 \& x_3 \& \bar{x}_5) \vee (x_1 \& \bar{x}_3 \& x_5),$$
$$x_{23} = x_{21} \vee x_{19} = (x_1 \& x_2 \& x_5 \& \bar{x}_3) \vee (\bar{x}_4 \& x_2 \& x_5)$$

and we get the diagram of Fig. 2.5,h.

Again, we eliminate the groups of series-connected contacts, this time using the expressions

$$x_{24} = x_{22} \& x_{17} = x_1 \& x_2 \& x_5,$$
$$x_{25} = x_{23} \& x_{11} = 0.$$

We thus obtain the circuit of Fig. 2.5,i. This last circuit can be represented by the function

$$x_e = x_{24} \vee x_{25} \vee x_3 = (x_1 \,\&\, x_2 \,\&\, x_5) \vee x_3.$$

This is the logical function performed by the original circuit of Fig. 2.5,a. But it is also performed by the circuit of Fig. 2.6; that is, the circuit shown in Fig. 2.6 is equivalent to that of Fig. 2.5,a.

Incidentally, contact x_4 of Fig. 2.5,a is absent from the equivalent circuit of Fig. 2.6; this means that it serves no purpose in the circuit, a fact which can be readily verified. Indeed, Fig. 2.5,a shows that the circuit can be closed by closing contact x_3 alone. If x_3 is open, then we can close the circuit by closing x_1, x_2, and x_5, and therefore do not need x_4.

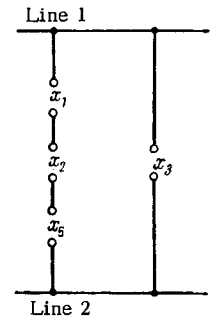


Fig. 2.6.

We have considered only one example of a procedure which allows us to derive the logical function corresponding to any given circuit. This procedure is called the *analysis* of the circuit. The simplification of the starting functions, arrived at by means of Boolean algebra, results in circuits that are equivalent to the starting networks but have the great advantage of being much simpler.

In our example we were able to simplify the function to such an extent that the practical switching circuit performing it could be drawn without further ado. This, however, is not always possible, especially in the more complex cases. In these cases, the logical function so derived is but the starting point in the *synthesis* of a switching network capable of realizing it.

2.3. SYNTHESIS OF COMBINATIONAL RELAY SWITCHING CIRCUITS

Assume that we are given some logical function and that we are required to construct a relay circuit embodying it. We shall discuss only circuits consisting of single-coil relays and an output relay whose coil is connected in series with the contact network. We shall assume that the logical function is given by a table enumerating all possible combinations of values of the variables, each such combination corresponding to some value of the function (such tables were described in Section 1.3).

Table 2.2

x_1	0	1	0	1	0	1	0	1
x_2	0	0	1	1	0	0	1	1
x_3	0	0	0	0	1	1	1	1
y	0	1	1	0	1	0	0	1

Consider the function of three variables $y = L(x_1, x_2, x_3)$, given by Table 2.2. Its complete disjunctive normal form (see Section 1.3) is:

$$y = (x_1 \& \bar{x}_2 \& \bar{x}_3) \vee (\bar{x}_1 \& x_2 \& \bar{x}_3) \vee (\bar{x}_1 \& \bar{x}_2 \& x_3) \vee (x_1 \& x_2 \& x_3).$$

This form is directly translatable into a practical switching network by means of the following rules:

As in the case of analysis, a normally open contact is made to correspond to x_i (in this particular function, $i = 1, 2, 3$), and a normally closed contact to $\bar{x}_i$. Each conjunctive term (in parentheses) becomes a chain of series-connected contacts, whose states are specified by variables contained in the parenthesis. The complete disjunctive normal form corresponds to parallel connection of the above-mentioned series chains.

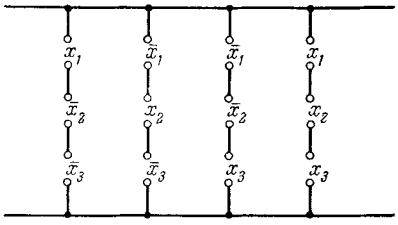


Fig. 2.7.

Applying these rules to our example, we obtain the circuit shown in Fig. 2.7.

This canonical technique will translate any logical function into a series-parallel switching network. In some cases it is then possible to simplify the result, obtaining a circuit containing a smaller number of elements (see the example of Section 2.2).

There are, however, other techniques, which by abandoning the canonical technique and the series-parallel network in favor of the so-called bridge circuit, lead directly to more economical systems embodying a given logical function.*

We shall now present, without proof, one such technique—devised by A. Sh. Blokh.** Returning to Table 2.2, which defined our logical function, let us write out the row containing the values of y, that

*However, these techniques still do not assure circuits with a minimum number of elements. The synthesis of circuits that are "minimum" with respect to some design variable is a serious problem for which there is, at present, no final solution. For a discussion, see Section 2.6.

**See [7], which contains all the necessary proofs. However, Blokh does not use the term "canonical" in the same sense as we do.

is

$$01101001.$$

We then group the symbols of this row into pairs as follows:

$$\underline{01}\ \underline{10}\ \underline{10}\ \underline{01}.$$

Should any pair contain two identical symbols, that symbol is written below that pair (there are no such pairs in this example). Otherwise, we assign the symbols 2 and 3 to the remaining pairs as follows:

$$\begin{matrix}01 & 10 & 10 & 01\\ \overline{2} & \overline{3} & \overline{3} & \overline{2}\end{matrix}.$$

These new symbols are again grouped into pairs; the new pairs are again grouped as above, and we assign the symbol 4 to the group 23 and the symbol 5 to group 32. The new symbols are again grouped, and so on, until a complete triangular matrix is obtained. This matrix will always have $k+1$ rows, where k is the number of arguments of the function. Thus the function of Table 2.2 yields the matrix

$$\begin{matrix}\underline{\begin{matrix}01 & 10\\ \overline{2} & \overline{3}\end{matrix}} & \underline{\begin{matrix}10 & 01\\ \overline{3} & \overline{2}\end{matrix}}\\ \underline{4} \qquad \underline{5}\\ \underline{6}\end{matrix}.$$

Now we proceed with the design of the switching network. To start with, we draw one horizontal line for each row of the matrix (in this case, $k+1=4$). We then enter each of the elements of the matrix as a point on the corresponding line, copying the respective numeral above that point; we thus obtain a set of nodes which we join into a "tree." In drawing the tree, we omit branches that lead to nodes denoted by 0 (see Fig. 2.8). On the branches originating in the lowest tier we place the contacts of the third relay (x_3 and $\bar{x}_3$), with $\bar{x}_3$ on the left- and x_3 on the right-hand branch. Similarly, contacts $\bar{x}_2$ and x_2 are positioned on the branches originating in the second lowest tier, while contacts $\bar{x}_1$ and x_1 are located on the branches starting from the third tier. If the tree contains a branch joining two nodes denoted by the same numeral, then the nodes are short-circuited (no contact is placed on the branch). For our example, we obtain the circuit shown in Fig. 2.9.

For all practical purposes, we now have a network performing the given function. This network may be simplified to its final form

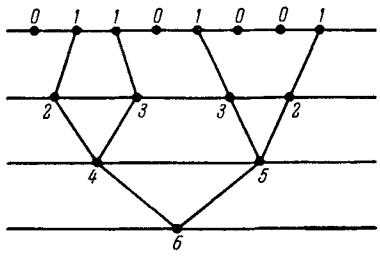


Fig. 2.8.

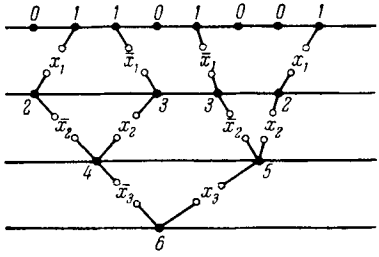


Fig. 2.9.

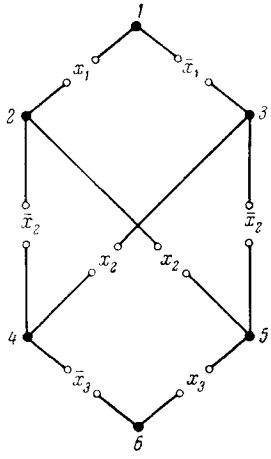


Fig. 2.10.

by combining the nodes denoted by the same numeral, thus reducing the number of component contacts. Our final design will then be the bridge circuit of Fig. 2.10.

The above technique yields a combinational relay switching circuit for any given logical function. This circuit usually contains a smaller number of elements than that synthesized by means of the canonical method employing the full normal disjunctive form of the function.

2.4. OTHER METHODS FOR CONVERTING LOGICAL FUNCTIONS INTO PRACTICAL DEVICES

Aside from electromechanical relays, there are other practical devices embodying logical functions, that is, capable of executing the operations of propositional calculus. We shall now give a few examples of these.

a) Diode Logic

A diode is an element with a nonlinear characteristic such that a flow (an electric current, a stream of air or of liquid, or any other flux) can pass through it in one direction virtually without resistance while a practically infinite resistance to this flow is offered in the opposite direction. Thus, the diode acts as a gate, allowing flow in one direction and blocking it in the other. In diagrams it is usually represented by the symbol shown in Fig. 2.11, where the triangle points in the allowed direction of flow.

In relay switching circuits, the input of the logical variables X_1, X_2, and so on, is accomplished by feeding current to the relay input. Wherever a negation (complement) of these variables is desired, one employs a normally closed contact. However, this cannot be done with diode circuits, because these circuits are incapable of performing the operation of negation.

Fig. 2.11.

For this reason, not only the variables x_1, x_2, and so on, but also their negations (complements) $\bar{x}_1$, $\bar{x}_2$, and so on, must be fed as inputs. These negations are performed outside the diode circuit by other devices, for instance, by electromechanical relays.

We shall now show how any logical function can be embodied in circuits employing only diodes and linear resistances. Let the function be given in its complete disjunctive normal form

$$y = y_1 \vee y_2 \vee y_3 \vee y_4 =$$
$$= (x_1 \& \bar{x}_2 \& \bar{x}_3) \vee (\bar{x}_1 \& x_2 \& \bar{x}_3) \vee (\bar{x}_1 \& \bar{x}_2 \& x_3) \vee (x_1 \& x_2 \& x_3).$$

This function has three independent variables and so the circuit must contain three pairs of lines—x_1 and $\bar{x}_1$, x_2 and $\bar{x}_2$, x_3 and $\bar{x}_3$. The number of output lines must equal the number of conjunctive terms (in parentheses) of the function being performed. In our case, there are four such terms (Fig. 2.12). All the output lines terminate in diodes whose terminals are, in turn, tied to a single output resistance. Such a circuit performs a disjunction in the same way as any other parallel connection. The input signals are also fed through resistances.

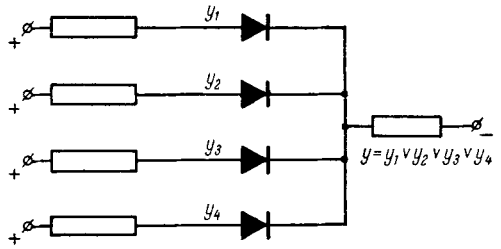


Fig. 2.12.

Each output line represents one of the conjunctive terms of our functional form. But the logical variables, at least in our case, are contained in all such parentheses. For this reason, each output line

must be connected, via diodes, with all those input lines which carry the variables contained in a given conjunctive term. The connecting diodes are arranged so as to permit the current to pass from the output to the input lines. In our example, the first conjunctive term is

$$y_1 = x_1 \& \bar{x}_2 \& \bar{x}_3 .$$

Its corresponding diode switching circuit is shown in Fig. 2.13; the complete diode circuit, performing the complete logical function, is shown in Fig. 2.14.

Any other logical functions may be performed in a similar fashion.

This technique starts from the complete disjunctive normal form of the function and is therefore as canonical a method as that employed for the synthesis of the relay switching circuits of Section 2.3. However, it usually yields circuits that are uneconomical because they require too many diodes. Although there are methods for designing more economical circuits, we shall not dwell on them here and shall refer the reader to the original publications (see, for example, [127]).

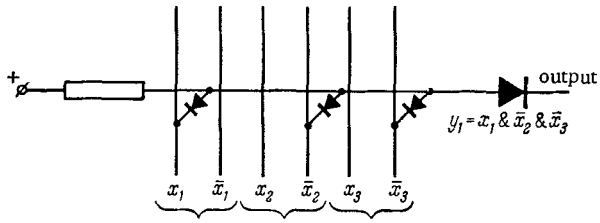


Fig. 2.13.

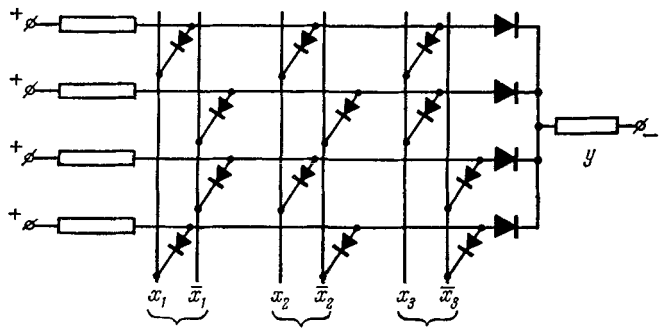


Fig. 2.14.

b) Triode Logic

A triode is an element exhibiting a variable resistance in response to a control signal. All other conditions being equal, the plate current of a triode (vacuum) tube, is determined by the grid voltage; in a transistor triode, the resistance varies as a function of an externally applied signal.

The characteristic curve of any triode device exhibits a saturation, at which the resistance is constant and maximum. We can use as logical variables (levels 0 and 1) the control signal levels which produce the minimum and maximum resistances of the triode. These resistances become the output of the device. Then various combinations of these triodes with constant passive resistances allow us to realize the logical functions of one (Fig. 2.15) and of several (Fig. 2.16) independent variables.

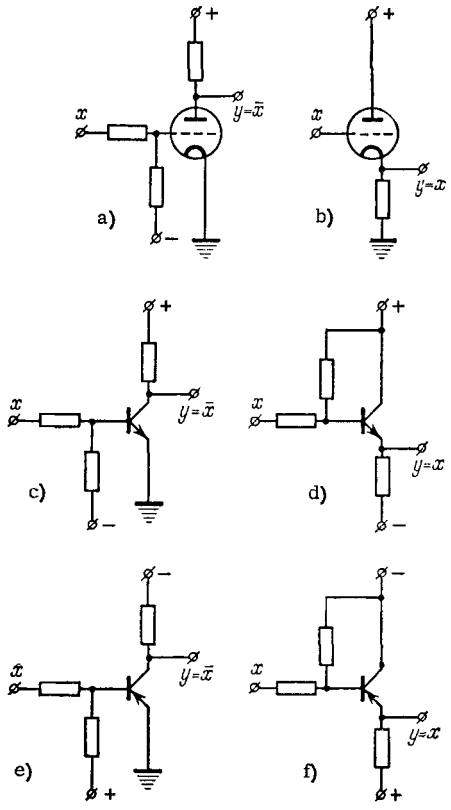


Fig. 2.15.

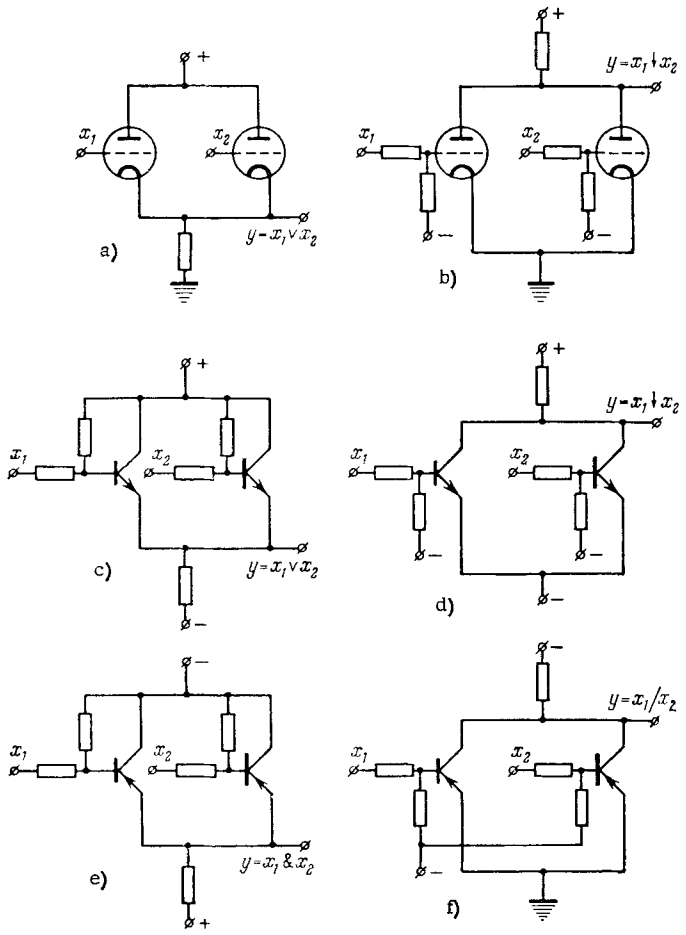


Fig. 2.16.

If one can execute negation, conjunction and disjunction by means of a set of devices, one can also perform any conjunctive term of the complete disjunctive normal form of any logical function, as well as the disjunction of these terms. This being the case, triode circuits can embody any logical function. Again, however, the canonical synthesis proves uneconomical (it yields circuits with redundant elements), so that one usually designs with more advanced techniques.

c) Networks using Magnetic Components

There are many ways of designing logical systems based on magnetic amplifiers, but we shall give only a brief description of a

greatly simplified version of one such system. Figure 2.17 is a schematic of a magnetic amplifier with positive feedback, consisting basically of a magnetic core *1* which is associated with several windings*. The alternating current is supplied to windings $w_{\sim}$ and $w'_{\sim}$, from which it passes, via the diode bridge *2* and the load resistance R_e, to the positive feedback windings w_{fb} and w'_{fb}. In addition, the core carries bias windings w_b and w'_b, as well as one or more control windings: $w_{con\,1}$ and $w'_{con\,1}$, $w_{con\,2}$ and $w'_{con\,2}$, and so on. The bias windings are supplied with a constant direct current i_b. The control windings are also dc-fed, and the levels of this direct current are used as the input variables of the system. The output of the device is the rectified current i_1 in the load circuit.

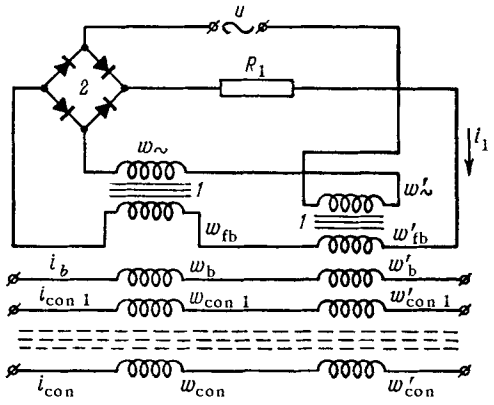


Fig. 2.17.

Consider first an amplifier with only one pair of control windings, $w_{con\,1}$ and $w'_{con\,1}$. Figure 2.18 shows the characteristic of this amplifier, that is, the dependence of the output (current i_1 in the load circuit) on the input (current i_{con} in the control windings) at zero bias current (a bias current shifts this characteristic along the i_{con} axis).

If the value $i_{con} = 0$ is made to correspond to the 0 level of the input variable and any value $i_{con} < -i'_{con}$ to the 1 level, and if the lowest and highest levels of the output current (these being the only possible levels, in accordance with the characteristic of Fig. 2.18) are made the 0 and 1 levels of the output variable, then, at zero bias current, the amplifier will be a negation element.

*The use of a split core and several pairs of windings eliminates ac pickup in the circuits carrying dc currents.

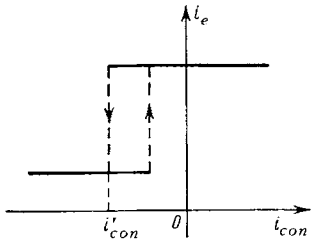


Fig. 2.18.

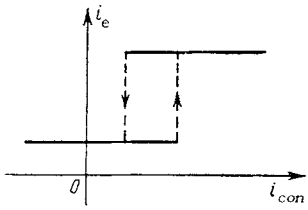


Fig. 2.19.

The same amplifier can perform "repetition." In this case, a bias current i_b is used to shift the characteristic to the right (as shown in Fig. 2.19), and the polarity of the control signal is reversed. Now the lowest level of output dc appears at $i_{con} = 0$; that is, the output logical variable is at level 0. If the input signal is 1 (i.e., i_{con} is high), then the level of the logical variable at the output will also be 1 (the current in the load circuit will be at maximum).

The magnetic amplifier is thus a contactless analog of the electromechanical relay with normally closed or normally open contacts.

We shall now consider a magnetic amplifier with several control windings. The characteristic of Fig. 2.19 will still hold at the appropriate bias current but the abscissa now denotes the total ampere-turns of all the control windings.

Retaining the same 0 and 1 levels of the individual input variables (that is, the same current values) in the corresponding windings as in the case of an amplifier with a single control winding, we now obtain a device performing a disjunction of all the n input variables. Indeed, it is now sufficient to set any one of the control windings at level 1 to obtain the maximum level of the output current.

If, however, the input current corresponding to level 1 in each winding is now reduced by a factor of n (where n is the number of input variables), then the number of ampere-turns necessary to obtain the same output level 1 can be achieved only if *all* the inputs are set equal to 1. The magnetic amplifier then embodies a conjunction of n variables and is the contactless analog of the multiple-coil electromechanical relay.

If the output of one magnetic amplifier is connected to the input of another magnetic amplifier (or to the inputs of several amplifiers), we have a network. In particular, we can use a set of these amplifiers to synthesize any desired combinational switching circuit. And since the individual magnetic components can perform negation, conjunction, and disjunction, a system of containing a multiplicity of such components can embody any desired logical function.

d) Pneumatically Operated Switching Circuits

A schematic diagram of a pneumatic switch is shown in Fig. 20.20,a and Fig. 2.20,b shows the conventional notation for it. The switch housing contains four chambers (K_1, K_2, K_3, and K_4) formed by the diaphragms M_1, M_2, and M_3 carried by a common piston rod R. Set-point controlling pressures P_a and P_b may be maintained in chambers K_1 and K_2 through ducts L_1 and L_2; chamber K_3 is connected to a compressed-air supply line via the axial duct C_3, and chamber K_4 is vented to the atmosphere via duct L_4. Axial duct C_4 from chamber K_4, and duct L_3 from chamber K_3 are interconnected on the outside by means of feedback line FB, in which we establish the output pressure P.

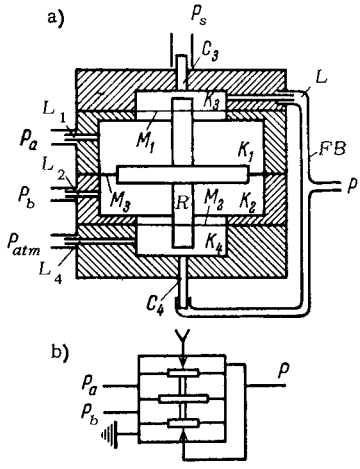


Fig. 2.20.

When the piston rod R is in its extreme "up" position, it blocks duct C_3 and opens up duct C_4; this produces atmospheric pressure in the FB line at the output of the switch. However, when the piston rod is in its extreme "down" position, it blocks C_4 and opens C_3; the output pressure then equals that in the supply line.

The position of the piston rod depends on the direction of the forces acting on its diaphragms, with the magnitude and direction of these forces determined by the pressures in chambers K_1, K_2, and K_3, that is, pressures P_a, P_b, and P. The opposing force exerted by the output pressure P on the diaphragm-rod assembly constitutes a positive feedback.

The response of this pneumatic switch, illustrating the above properties, is presented in Fig. 2.21.

Now consider this switch when a constant pressure $P_b = P_{h1}$ (the back pressure, or bias pressure) is maintained in K_2. The shape of the response remains unchanged from that of Fig. 2.21, but it is displaced to the right, the magnitude of the displacement increasing with back pressure (bias) P_b (Fig. 2.22).

Such a device can be used to perform the logical operation of "repetition." This is done by assigning the level 1 to a pressure higher than P_{h1}, and the level 0 to a pressure lower than $(P_{h1} - \Delta P)$.

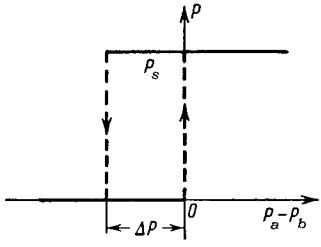


Fig. 2.21.

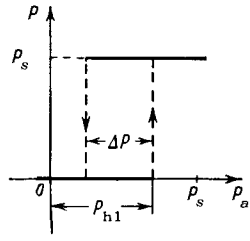


Fig. 2.22.

Obviously, the supply pressure P_s is that exceeding P_{h1}. The switch performing the repetition is shown in Fig. 2.23 in the conventional notation of Fig. 2.20, with the chamber in which pressure P_{h1} (first back pressure) is maintained, indicated by cross-hatching.

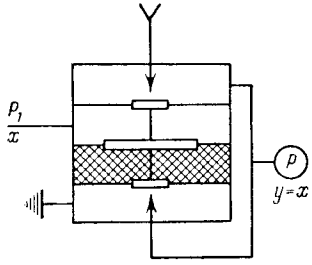


Fig. 2.23.

The pneumatic switch may also be used to perform negation. In this case a constant pressure P_{h2} is maintained in K_1, and the response of the switch is shown in Fig. 2.24. The simplified diagram is shown in Fig. 2.25, with K_1, in which the constant pressure P_{h2} (differing from P_{h1}) is maintained, indicated by hatching. Pressures P_{h1} and P_{h2} differ because P_{h1} determines the location of the right-hand and P_{h2} that of the left-hand vertical line of the hysteresis loop. Our device now performs a negation. Thus we make the independent logical variable $P_1 = 1$ at $P_b > P_{h2} + \Delta P$; the output signal then assumes level 0; if $P_b < P_{h2}$ (that is, when $P_1 = 0$), the output pressure $P = P_s$, and the output signal equals 1.

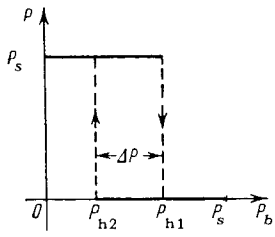


Fig. 2.24.

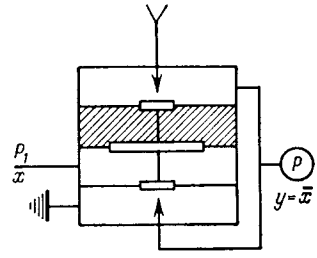


Fig. 2.25.

So far, we have shown how the switch performs logical functions of one independent variable. We shall now show how the same switch can perform logical functions of two or more independent variables.

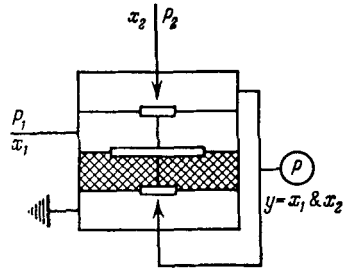


Fig. 2.26.

The schematic diagram of Fig. 2.26 shows that the duct previously leading to the supply line (Fig. 2.23) is now connected to the line producing a second independent input variable P_2. The switch will now perform the conjunction of two independent variables because an above-atmospheric pressure will exist in the output line if, and only if, both input signals are at level 1.

A circuit of $n-1$ devices, assembled as in Fig. 2.27, will perform the conjunction of n independent variables.

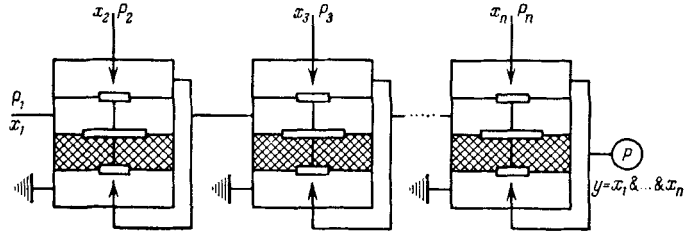


Fig. 2.27.

Figure 2.28 shows a device performing the disjunction of two independent variables while the circuit of Fig. 2.29, which consists of $(n-1)$ pneumatic switches, performs the disjunction of n independent variables.

Since we now have pneumatic devices performing negation, conjunction, and disjunction, we can design pneumatic switching circuits to perform any logical function. Here, too, the canonical method may be used (as we have already stated several times, this method starts with a given function in its complete disjunctive normal form). But, as before, this general procedure yields switching circuits that

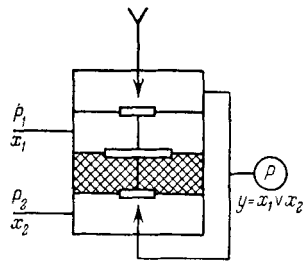


Fig. 2.28.

are uneconomical because they require too many components. We can see this from the mere fact that our pneumatic switch, which may be employed as a device performing negation, repetition, conjunction, and disjunction of two independent variables, can also be used as an implication (Fig. 2.30) or as an inhibit device (Fig. 2.31). The figure shows that implication can be achieved by means of a single switch, whereas the canonical method, which expresses implication by means of negation, conjunction, and disjunction, calls for two such devices. This follows from the formula

$$P = P_1 \to P_2 = \bar{P}_1 \vee P_2.$$

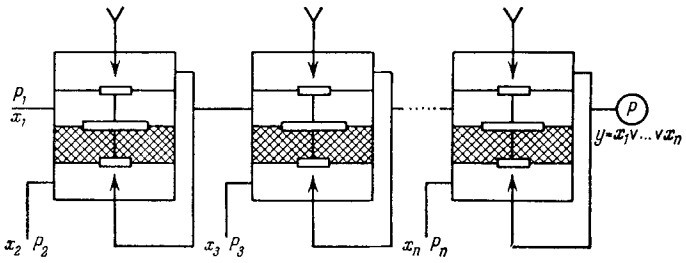


Fig. 2.29.

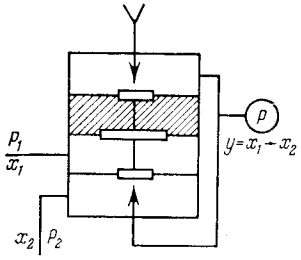


Fig. 2.30.

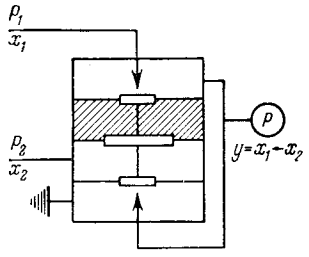


Fig. 2.31.

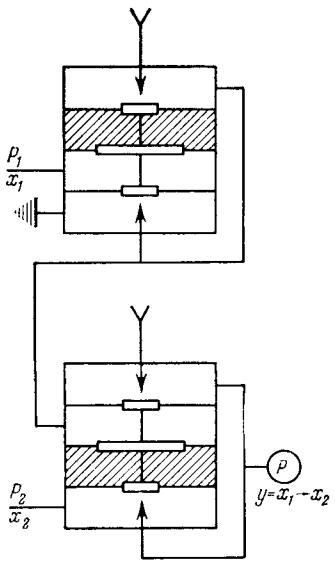


Fig. 2.32.

A switching circuit performing implication as required by the canonical method is shown in Fig. 2.32. This cumbersome arrangement performs the same function as the simple switch of Fig. 2.30.

2.5. THE PROBLEM OF MINIMIZATION OF DEVICES PERFORMING LOGICAL FUNCTIONS

The design of devices performing given functions immediately entails the following problem: Given a set of blocks (elements) capable of performing simple logical functions, each kind of block being associated with some positive number called its "price" (this may be the actual price or some conventional factor), and given also the function to be executed (for example, in its full normal disjunctive form), we want to determine which of the switching circuits capable of performing this function (and consisting of the blocks of the given set) will have the minimum total price, defined as

$$p = \sum_{i=1}^{r} \alpha_i h_i,$$

where α_i is the number of blocks of a particular kind, h_i is the price of one block, and r is the number of different blocks in the set.

This problem, often referred to as the minimization problem, is of fundamental importance in engineering applications of propositional calculus. A great deal of work has been devoted to it, and numerous algorithms* (procedures) suggested for its partial solution. All these procedures consists of more or less complex scanning methods (that is, examination of all the different existing possibilities), so that so far there are no convenient, practical techniques for minimization; all that has been developed is various "paths" along which one may hope to come across more or less economical designs.

To give the reader at least some broad idea of what is involved, we shall briefly describe one of the many procedures for partial solution of the minimization problem.

Suppose the logical function F is given in its complete disjunctive normal form. If the set of blocks consists of the AND, OR, and NOT elements (both AND and OR having two inputs each) and all the elements carry the same price tags, then the minimization problem is reduced to finding that analytical expression of this function which

*The term "algorithm," translated here for convenience as procedure, shall be frequently encountered in subsequent chapters. It shall be defined in Chapter 7.

contains only the symbols $\bar{}$, &, and $\vee$ and in which the number of these symbols is minimum.

Let us describe Quine's solution of this problem [214]. The full form is simplified as much as possible by means of the identity

$$(A \& x) \vee (A \& \bar{x}) = A, \tag{2.1}$$

where A may be a conjunction of several variables. Then the same operation is repeated on all the conjunctions obtained as a result of the first simplification, and so on, until no further reduction of terms is possible. The pairs of conjunctions (originating from the terms of the complete form and from the form obtained as a result of the simplification) which cannot be further reduced by means of the simplifying identity (2.1) are called the *prime implicants of* F. Quine has shown that any minimal disjunctive normal expression of F is a disjunction of certain prime implicants of F. Therefore, the next step in finding the minimal expressions of F consists of determining these combinations of prime implicants whose disjunctions yield minimal expressions. This technique (see [185]) gives combinations of prime implicants whose disjunction is equivalent to F, but from which not a single prime implicant may be eliminated without violating the condition of equivalence to F. Such disjunctions are called "irredundant" expressions for F. Then we count the number of symbols $\bar{}$, &, and $\vee$ in each of the irredundant expressions and select the expressions with the least total of these symbols. These are the minimal expressions, according to our criterion of minimality.

Consider an example. We are given the Boolean function

$$F(x_1, x_2, x_3) = (\bar{x}_1 \& x_2 \& x_3) \vee (x_1 \& \bar{x}_2 \& x_3) \vee \\ \vee (x_1 \& x_2 \& \bar{x}_3) \vee (\bar{x}_1 \& \bar{x}_2 \& x_3) \vee (\bar{x}_1 \& x_2 \& \bar{x}_3).$$

As a result of all possible pairwise reductions of terms of the complete form (whereby each term of the disjunctive form may be used in more than one pair), we obtain the conjunctions

$$\bar{x}_1 \& x_3, \quad \bar{x}_1 \& x_2, \quad \bar{x}_2 \& x_3, \quad x_2 \& \bar{x}_3,$$

which cannot be further reduced. These are also the only conjunctions whose combination does not yield a single further reduction. Thus they all are prime implicants of F. Although the disjunction of all these prime implicants is equivalent to F, it may be proven directly that the deletion of the conjunction $\bar{x}_1 \& x_3$ does not violate the condition for equivalence but that no other remaining conjunction can be deleted without violating that equivalence. Hence,

$$F = (\bar{x}_1 \& x_2) \vee (\bar{x}_2 \& x_3) \vee (x_2 \& \bar{x}_3)$$

is one of the irredundant expressions. It may also be shown that

$$(\bar{x}_1 \& x_3) \vee (\bar{x}_2 \& x_3) \vee (x_2 \& \bar{x}_3)$$

is also an irredundant expression. The function has no other such expressions. A comparison of these two irredundant expressions shows that they have the same number of $\bar{}$, &, and $\vee$ symbols; therefore they are minimal to the same degree.

So much for Quine's procedure. We now have dozens of procedures for finding prime implicants of logical functions. Some of these are more suited for manual calculations, others for computations on computers; still others are mainly employed in research on minimization problems. The methods of minimization also differ: one can use special diagrams [180], various constructs on n-dimensional cubes [33], numerical calculations [161, 127], and so on.

Several procedures (for example, [33]) develop minimal normal expressions by starting with the prime implicants.

The finding of minimal normal expressions of logical functions of even a small number of variables (for instance, six or seven) is a rather laborious process. But we now have several useful simplified procedures which give normal expressions that are close to the minimal and entail much less labor [216-218].

The Quine procedure yields minimal disjunctive normal expressions. However, the minimal conjunctive normal expressions may sometimes prove to be "smaller" than the disjunctive forms. Because of that, one must examine both the disjunctive and the conjunctive normal expressions to select the truly minimal expression. Since the techniques for obtaining minimal conjunctive normal expressions are similar to those for the corresponding disjunctive forms, we shall not dwell on them.

The fact that a function yields a minimal normal expression does not necessarily mean that an even simpler expression cannot be obtained. For example, the minimal disjunctive normal expression of the function

$$F(x_1, \ldots, x_6) = (x_2 \& x_3 \& x_4 \& x_6) \vee (x_1 \& x_2 \& x_5) \vee \vee (x_1 \& x_2 \& x_6) \vee (x_1 \& x_3 \& x_4 \& x_5) \tag{2.2}$$

has thirteen & and $\vee$ symbols, whereas the minimal conjunctive normal expression of the same function

$$F(x_1, \ldots, x_6) = (\bar{x}_1 \& \bar{x}_3) \vee (\bar{x}_1 \& \bar{x}_4) \vee (\bar{x}_2 \& \bar{x}_3) \vee (\bar{x}_2 \& \bar{x}_4) \vee \vee (\bar{x}_1 \& \bar{x}_2) \vee (\bar{x}_1 \& \bar{x}_6) \vee (\bar{x}_2 \& \bar{x}_5) \vee (\bar{x}_5 \& \bar{x}_6)$$

contains $8 + 7 + 16 = 31$ $\bar{}$, &, and $\vee$, symbols; thus, (2.2) is the minimal normal expression. But another expression for the same function

$$F(x_1, \ldots, x_6) = [x_1 \& x_2 \& (x_5 \vee x_6)] \vee \{x_3 \& x_4 \& [(x_1 \& x_5) \vee (x_2 \& x_6)]\} \quad (2.3)$$

contains only nine & and $\vee$ symbols.

In this case we have reduced the minimal normal expression by means of the identity $(A\&B) \vee (A\&C) = A\&(B \vee C)$. Sometimes, however, one can employ this distributive law to the hilt and still not come up with the real minimum expression. For example, our minimal normal function (2.2) can be reduced still further

$$F(x_1, \ldots, x_6) = [(x_1 \& x_2) \vee (x_3 \& x_4)] \& [(x_1 \& x_5) \vee (x_2 \& x_6)].$$

This form may be obtained from (2.3) by expanding the first term of the disjunction

$$x_1 \& x_2 \& (x_5 \vee x_6) = x_1 \& x_2 \& [(x_1 \& x_5) \vee (x_2 \& x_6)],$$

and employing the distributive law to reduce the new expression.

Obviously, the reduction of other functions requires other identities. It is very difficult, however, to select *a priori* an identity suitable for the reduction of a given expression. In fact, it is even difficult to say *a priori* whether a given expression can be eventually reduced to a more manageable form. Thus a great forward step would be a procedure yielding expressions about which one could confidently say that they are as "small" as can be found, that is, that there are no other forms of a given function which are more "minimal" [120, 121]. Such expressions are called *absolutely minimal*, and their finding involves procedures which are far more complex than those for minimal normal expressions. We shall therefore not discuss them in detail, but shall simply point out that each such nontrivial procedure (algorithm) should have the following two features:

1. It should be able to predict the maximum complexity associated with the absolutely minimal expressions of a given function.
2. It should be able to give the absolutely minimal expressions within the limits imposed by the predicted maximum complexity.

For example, one can predict that the absolutely minimal expressions for the function (2.2) are not more complex than a "disjunction of conjunctions of disjunctions" (type I) and a "conjunction of disjunctions of conjunctions" (type II). One then uses special algorithms to express (2.2) in terms of these limiting forms I and II. This gives two expressions of type I

$$F(x_1, \ldots, x_6) = [x_3 \& (\bar{x}_1 \vee \bar{x}_2)] \vee (x_2 \& \bar{x}_3),$$
$$F(x_1, \ldots, x_6) = [x_2 \& (\bar{x}_1 \vee \bar{x}_3)] \vee (\bar{x}_2 \& x_3)$$

and one "degenerate" expression of type II

$$F(x_1, \ldots, x_6) = (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \& (x_2 \vee x_3).$$

Notice that both expressions resemble irredundant forms. With only three forms to scan, it is easy to see which is the smallest.

In general, however, the number of expressions similar to the irredundant forms is extremely large, even if the function has only a few variables, and the above procedures for absolutely minimal expressions are therefore not practical. For this reason the problem was attacked by developing procedures involving considerably fewer elementary operations. Such procedures necessarily yield expressions of a more complex form than the normal, but such expressions in general tend to approach the absolutely minimal. For example, the procedure may involve successive applications of the distributive law to the prime implicants of a given function. The resulting complex implicants may then themselves be treated as prime implicants and serve as a basis for developing irredundant expressions. The final minimal irredundant expressions can then be selected from these irredundant forms in the usual way.

However, there is another problem. Even if the absolutely minimal expression is known, the circuit based on it may prove to be nonminimal. For example, the absolutely minimal expression of the function

$$F = x_3 \& [x_5 \& (x_1 \vee x_2) \vee (x_2 \& x_4)] \vee \vee [x_6 \& [(x_1 \& x_5) \vee (x_2 \& x_4)]], \tag{2.4}$$

immediately yields a switching circuit of ten elements. However, a circuit performing this same function can also be constructed from eight elements (Fig. 2.33). This is due to the fact that, in some cases, one section or block of a system can be used to embody more than one part of the minimal expression. Thus we can represent

(2.4) in the form

$$F = \{x_3 \& [(x_5 \& x_1) \vee (x_5 \& x_2) \vee (x_2 \& x_4)]\} \vee [x_6 \& [(x_1 \& x_5) \vee (x_2 \& x_4)]] = \{x_3 \& [Q \vee (x_5 \& x_2)]\} \vee (x_6 \& Q)$$

where

$$Q = (x_1 \& x_5) \vee (x_2 \& x_4), \tag{2.5}$$

Our actual circuit of Fig. 2.33 can then be reduced to eight elements because the Q operation, which appears twice in the irredundant expression, can be iterated through one and the same circuit block.

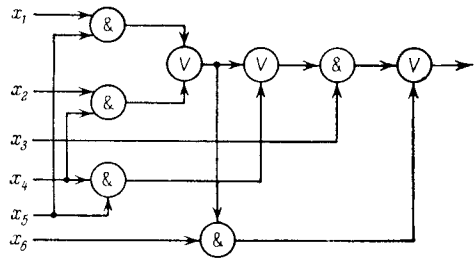


Fig. 2.33.

We have briefly reviewed the minimization problem assuming that all the elements carry the same price tag. It has been shown [217] that the minimization of circuits comprising elements of differing prices can be achieved by modifications of the same methods. The only difference is that a different criterion of the minimum is used in selecting the minimal expressions from the irredundant forms.

Our discussion was confined to minimization of sets consisting only of NOT, AND, and OR blocks, where the AND and OR units had only two inputs each. There are also solutions for similar problems involving other sets. However, each new set requires a new solution of the minimization problem. Thus, if the set consists of blocks of negation as well as of conjunction and disjunction of n variables, the problem reduces to finding irredundant expressions (or expressions similar to irredundant forms if we deal with compound expressions) in which the number of prime implicants is minimum.

The minimization problem has become especially important due to the advent of general-purpose elements, that is, blocks that, either by means of simpler readjustment or by adding external connections

which cost little or nothing, may be used to perform several different functions. A typical example of such a block is the pneumatic switch of Fig. 2.20. There are no accepted solutions of the minimization problem for these systems, despite many attempts at developing one. The present trend in these systems is to develop procedures which would yield circuits that, while not minimal, are sufficiently minimized for practical purposes.

3

Finite Automata and Sequential Machines: Basic Concepts

3.1. DISCRETE TIME AND DISCRETE TIME MOMENTS

Let $\{x\}_i$ $(i = 1, 2, \ldots, n)$ and $\{y\}$ be alphabets containing a finite number of symbols. Then the functional relationship

$$y = f(x_1, x_2, \ldots, x_n) \tag{3.1}$$

matches any set of symbols, taken one at a time from alphabets $\{x\}_1, \{x\}_2, \ldots, \{x\}_n$, with one symbol of alphabet $\{y\}$.

Now consider an ideal device embodying relationship (3.1). This device has n inputs and a single output. Inputs $x_1, x_2, \ldots, x_n$ are fed symbols from alphabets $\{x\}_1, \{x\}_2, \ldots, \{x\}_n$, respectively, all these inputs being made in a block, that is, at exactly the same time. This instantaneously generates a symbol from alphabet $\{y\}$ at the output, as specified by (3.1). We shall call such an instantaneously operating ideal device a *function converter.* In the special case when each of the alphabets $\{x\}_1, \{x\}_2, \ldots, \{x\}_n$ and $\{y\}$ consists of two symbols only, that is, when $x_1, x_2, \ldots, x_n$ and y are logical variables and f is a logical function, such a device is a *logical converter.* Instantaneously responding combinational relay switching circuits and similar devices for performing the operations of propositional calculus would be practical embodiments of the abstract concept of a "logical converter."

So far, our functional relationships have neglected the time factor and we have also assumed that the function converter acts instantaneously. Now, however, we shall introduce the concept of time.

We usually assume that time varies only in one direction ("into the future"), that it varies continuously, and that it thus passes through all possible values on the positive real axis. In other words,

when time appears as an argument of a function, it is usually defined on a continuum, namely, the positive real axis (the time axis).

In contrast to this, it is convenient to study discrete-action devices in terms of a hypothetical discrete time. Let us imagine that the continuous time axis can be divided into an infinite number of finite intervals, not necessarily of equal length (Fig. 3.1). Moving along the axis from $t = 0$ toward $t = \infty$, we mark the points separating these intervals by characters $t_0, t_1, t_2, \ldots$ These points then constitute a countable set.

Let us further agree to represent the characters $t_0, t_1, t_2, \ldots$ by a series of positive integers 0, 1, 2, ... and call that imaginary time which consecutively assumes only these integral values the *discrete time* t.

The time instants $t_0, t_1, t_2, \ldots$, now denoted by numbers 0, 1, 2, ..., shall be called *discrete moments*, and the numbers 0, 1, 2, ... shall be treated as symbols constituting an alphabet $\{t\}$.

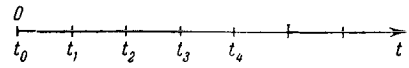


Fig. 3.1.

The current discrete moment (the one corresponding to the present instant) shall be denoted by p (present). Thus p divides all t's into those preceding p ($p-1, p-2, \ldots$) and those following ($p+1$, $p+2, \ldots$) [Fig. 3.2].

Thus far, we have treated the variables of Eq. (3.1) as time-variant. Assume now that $x_1, x_2, \ldots, x_n$ vary in the discrete time. That is, the variables assume definite values for each t, so that we have functions $x_i(t)$, where $t = 0, 1, 2, \ldots$, and where $\{x\}_i$ assumes values from alphabets $\{x\}_i$ ($i = 1, 2, \ldots n$). Then, by virtue of (3.1), we can set up a correspondence between these functions and the function

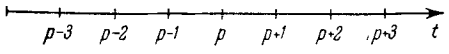


Fig. 3.2.

$$y(t) = f[x_1(t), x_2(t), \ldots, x_n(t)], \tag{3.2}$$

where y varies with the same t as x_i and assumes values from alphabet $\{y\}$. Such a system operates in discrete time but "has no memory" in the sense that the "output" value y at any instant $t = p$ depends solely on the values of the "inputs" x_i at that instant.

One can, however, imagine systems which also operate in discrete time and whose inputs and outputs are also symbols drawn from infinite alphabets, but in which the relationship between the

input and the output is not so simple. For example,

$$y(p) = f[x_1(p-1),\ x_2(p-1),\ \ldots,\ x_n(p-1);\ x_1(p),\ x_2(p),\ \ldots,\ x_n(p)]. \tag{3.3}$$

In other words, the value of y at any $t = p$ depends not only on the values of all the x_i at p but also on their values of the preceding discrete moment $p-1$. Further, the value of y at $t = p$ may even be a function of the entire history of values y. Consider the case in which y is a logical variable whose value at any p is specified as a negation of the value of y at the preceding moment $p-1$:

$$y(p) = \overline{y(p-1)}. \tag{3.4}$$

Although the relation

$$y = \bar{y}$$

is contradictory, Eq. (3.4) does not lead to contradictions; it specifies a function $y(t)$ which consecutively assumes the values 1 and 0 even though there are no input (external) signals.

These special dynamical systems differ from the common ones (such as the pendulum or the four-terminal network) in that they operate in discrete time and their coordinates (inputs and outputs) are defined on finite sets.

Henceforth we shall be dealing with dynamical systems that are distinguished by these two properties.

3.2. ON DYNAMICAL SYSTEMS

A *dynamical system* is one involving time-varying processes. The state of a natural or man-made dynamical system at any instant is given by some number (finite or infinite) of *generalized coordinates*. Dynamical systems may be divided into several classes, depending on:

a) whether they are time-continuous or time-discrete, that is, whether the time is assumed to vary in a continuum or a countable set;

b) whether the system has a finite or infinite number of generalized coordinates; and finally

c) the cardinality of the set of all possible values of each generalized coordinate, that is, whether these sets are finite, countably infinite, or continua.

The concept "dynamical system" is usually associated with systems described by ordinary or partial differential equations. In systems of this type the number of generalized coordinates may be finite (in which case they are described by ordinary differential equations) or infinite (described by partial differential equations), but in either case both the coordinates and the time vary in continua.

In those cases where the time is discrete, that is, varies in a countable set, while each of the finite or infinite number of generalized coordinates may assume values from continuum sets, the behavior of the system is described by difference equations.

In a special class of dynamical systems the time is again discrete but the generalized coordinates (whose number may be finite or infinite) assume values from finite sets.

Every dynamical system may be affected by externally generated input signals. Such input signals may also be defined on a continuum, a countable set, or a finite set. Dynamical systems described by differential or difference equations are usually capable of handling only a finite number of input signals; the latter, however, may take on any values from some continuum. Dynamical systems whose generalized coordinates are defined on finite sets are usually analyzed in terms of a finite number of input signals, and each of these signals is also defined on a finite set.

Dynamical systems in which time is defined on a countable set, the coordinates and (externally generated) input signals are defined on finite sets, and the number of input signals and coordinates is finite will be called *finite dynamical systems*. Particular cases of this class of systems are finite automata and sequential machines.

Systems that differ from the finite only in that they have an infinite number of generalized coordinates constitute a more general class of dynamical systems. These include Turing machines* and similar idealized devices.

The reader must be reminded at this point that an equation describes only an idealized model and not a real system. In this sense any dynamical system is an abstraction. But although finite dynamical systems and Turing machines are no more than abstractions, they are very important abstractions because many technical devices and important natural processes lend themselves to representation by such abstractions.

*See Chapter 8.

3.3. FINITE AUTOMATA

Consider a finite dynamical system whose state at any instant is characterized by a finite number of generalized coordinates $\varkappa_1, \varkappa_2, \ldots, \varkappa_n$. This system is subject to a finite number of externally generated input signals $\rho_1(t), \rho_2(t), \ldots\ldots, \rho_m(t)$. We are given either a time scale divided into discrete moments or conditions that enable us to determine the instant at which the next discrete moment will occur. In our definition, the signals and the states of the system are meaningful *only* at such discrete moments (and are neglected at all other times).

At these moments, each of the generalized coordinates $\varkappa_i$ may take on values only from a finite set, while each input ρ_j also assumes values only from its finite set (of input signals).

Let us introduce an n-dimensional vector $\varkappa$ with coordinates $\varkappa_1, \varkappa_2, \ldots, \varkappa_n$ and an m-dimensional vector ρ with coordinates $\rho_1, \rho_2, \ldots, \rho_m$. Because all the coordinates of the vector $\varkappa$, that is, $\varkappa_1, \varkappa_2, \ldots, \varkappa_n$ are defined on finite sets, the vector $\varkappa$ is also defined on a finite set. If the coordinate $\varkappa_i$ may take on ξ_i values, then the vector $\varkappa$ may assume one of $k = \prod_{i=1}^{n} \xi_i$ values. Accordingly, the set on which $\varkappa$ is defined consists of k elements.

By exact analogy, the vector ρ with coordinates $\rho_1, \rho_2, \ldots, \rho_m$ is given on a finite set containing r elements, where $r = \prod_{j=1}^{m} \eta_j$, and η_j is the number of elements in the set on which ρ_j is defined.

Let us consider an alphabet $\{\varkappa\} = \{\varkappa_1, \varkappa_2, \ldots, \varkappa_k\}$ consisting of k symbols, and let us match the various possible values of the vector $\varkappa$ to the various symbols from this alphabet. We shall call the vector $\varkappa$ the *state* of our finite dynamical system.

Similarly, let us introduce an alphabet

$$\{\rho\} = \{\rho_1, \rho_2, \ldots, \rho_r\},$$

consisting of r symbols, and match the various values of the vector ρ to the various symbols from this alphabet. We shall call the vector ρ the *input* to the system.

Now we shall define "motion" in our system, that is, we shall specify the method by which the state of the system is defined at each discrete moment of time. One very important definition leads to the concept a finite automation.

Definition. A finite dynamical system is said to be a finite automaton if its state at each discrete moment is uniquely defined (a) by

its state in the preceding moment and (b) by its input at the preceding or the current moment.

A finite automaton whose current state is defined by its state and input at the preceding moment shall be called a *finite automaton of the P - P* (past — past) *type.* An automaton whose current state is defined by its state at the preceding moment and its current input shall be called a *finite automaton of the P - Pr* (past-present) *type.*

The term "finite automaton" also includes finite systems whose states are defined by their states and inputs during any desired finite number of preceding moments. The term does not, however, pertain to finite systems whose states are defined by random factors or by their entire history (that is, systems whose states and inputs cannot be specified unless one knows their value at all the preceding discrete moments).

To put the above definition of the finite automaton in other terms, the symbol $\varkappa$ at any discrete moment is uniquely defined by the $\varkappa$ of the preceding moment and ρ at the preceding or in the current moment. That is, for a P - P automaton:

$$\varkappa^{p+1} = F(\varkappa^p, \rho^p), \quad p = 0, 1, 2, \ldots, \tag{3.5'}$$

and for a finite automation of the P - Pr type

$$\varkappa^{p+1} = F(\varkappa^p, \rho^{p+1}), \quad p = 0, 1, 2, \ldots, \tag{3.5''}$$

where F is a function in the sense of Chapter 1 (it matches a symbol from the alphabet $\{\varkappa\}$ with symbols from the alphabets $\{\varkappa\}$ and $\{\rho\}$. However, in contrast to the sense of Chapter 1, the symbol-arguments and the symbol-function may now pertain to differing time instants. For this reason formulas (3.5) do not specify a converter but a dynamical system.

The discrete moments corresponding to given $\varkappa$ and ρ are identified by superscripts, where p stands for the present, $p + 1$ the next, and $p - 1$ the immediately preceding moment.

If a new symbol μ is defined in the same alphabet $\{\varkappa\} = \{\varkappa_1, \varkappa_2, \ldots, \varkappa_k\}$ as $\varkappa$, then relations (3.5′) and (3.5″) can be treated as derived from

$$\left.\begin{aligned} \mu &= F(\varkappa, \rho), \\ \varkappa^{p+1} &= \mu^p. \end{aligned}\right\} \tag{3.6}$$

In the first of these relations, all the symbols pertain to the same time. If that time is p, that is,

$$\mu^p = F(\varkappa^p, \rho^p),$$

when we add the second relations (3.6) and eliminate μ^p we get

$$\varkappa^{p+1} = F(\varkappa^p, \rho^p),$$

that is, relation (3.5′).

If, however, the time corresponding to the first of relations (3.6) is $p + 1$, that is

$$\mu^{p+1} = F(\varkappa^{p+1}, \rho^{p+1}),$$

we can add the second of relations (3.6), eliminate $\varkappa^{p+1}$ and get

$$\mu^{p+1} = F(\mu^p, \rho^{p+1}),$$

that is, relation (3.5″).

Let us consider (3.5′):

$$\varkappa^{p+1} = F(\varkappa^p, \rho^p).$$

Knowing ρ^0 and $\varkappa^0$ for the moment zero, we can, by putting $p = 1$, find $\varkappa^1$. Then, knowing $\varkappa^1$ and ρ^1, we can find $\varkappa^2$, and so on. The values of $\varkappa^1, \varkappa^2, \ldots$ can be determined in a similar fashion from Eq. (3.5″), starting from a given $\varkappa^0$ and a given input sequence $\rho^1, \rho^2, \ldots$ In this respect formulas (3.5) determine recurrence relations, which permit us to find consecutively all the $\varkappa^1, \varkappa^2, \ldots$, provided the initial state $\varkappa^0$ and the input sequence $\rho^0, \rho^1, \rho^2, \ldots$ are known.

We have already stated that $\varkappa$ and ρ as well as the behavior of the system, in general, are significant only during discrete time moments. Thus, in dealing with a real dynamical system (or process) we imagine a device that records (samples) the values of $\varkappa$ and ρ at such moments*, and that the decision as to whether or not the system is a finite automaton is made only on the strength of such a sampling record. In this sense the abstract concept of a "finite automaton" may also be employed to describe continuous devices (devices exhibiting a continuum of states varying in

*Or that there exists a stroboscopic device illuminating the observed process only at these instants.

continuous time), provided only that during the discrete sampling moments, when the system is observed, the set of its possible states is finite and that one of relations (3.5) is satisfied. Thus, for example, a continuous system having several equilibrium states may be treated as a finite automaton. This is possible provided the data sampling moments are made to coincide with the instants at which complete equilibrium becomes established and provided the state of equilibrium is always uniquely determined by the system's preceding equilibrium and by the input signals to which it is subjected at the instant when that equilibrium is disturbed (or established).

Since all real systems operate in continuous time, the use of discrete time in our discussion calls for a special device, a synchronizing source, which signals the advent of the next discrete time moment (that is, data sampling moment). We shall call such a source a ***discrete clock*** (or simply a clock).

The clock is not an integral part of the finite automaton. The signals it generates are external to the automaton in the same sense as are the input signals ρ. But the clock signals—the time input—do differ from the externally generated input signals, since they are not coded in symbols from the alphabet $\{\rho\}$ and they do not constitute arguments of function F in (3.5). If the finite automaton is a process, the clock signals can be used only in some device that records ρ and $\varkappa$ at the various time instants. In technical embodiments of a finite automaton, the clock signals are used only to determine the advent of the next discrete time moment.

Let us now examine some examples of the division of a continuous time scale into discrete moments.

a) The continuous time is divided into equal intervals so that an ordinary clock with a suitable regulated movement may serve as a synchronizing source. This is *uniform* time division.

b) The next discrete moment occurs wherever the symbol ρ is changed, that is, whenever there is a change in the input. Here, the continuous time is divided into a sequence of intervals that are not, in general, of the same length. The clock may then be any device that responds to a change in input.

c) The next moment occurs whenever a symbol ρ_i or ρ_j appears at the input.

d) The next moment occurs whenever a symbol ρ with an odd superscript is replaced by a ρ with an even superscript; and so on.

Returning to formulas (3.5), let us now assume that the input ρ does not vary. Then we have

$$\varkappa^{p+1} = F[\varkappa^p, \rho_*], \tag{3.7}$$

where ρ_* is a constant value of ρ. We shall call such a finite automaton *self-contained* or *autonomous*. It is independent of the externally generated input signals, but it still employs clock signals to indicate the next discrete time moment.

The symbol $\rho = \rho_*$ may be regarded as a parameter because it may be assigned any symbol from the alphabet $\{\rho\}$. By so doing we obtain r autonomous automata. In this sense each finite automaton may be transformed into r autonomous automata (some of which may be identical).

There is still another, formal definition of a finite automaton. This definition is unrelated to the concepts of a finite dynamical system, a state or an input. It merely says that *given two finite alphabets of symbols* $\{\varkappa\}$ *and* $\{\rho\}$, *as well as the variables* $\varkappa$ *and* ρ *which assume values from these alphabets, a finite automaton consists of the recurrence relations (3.5) coupling these variables.*

This is a very broad and a very abstract definition, but its value lies precisely in its generality. It applies to a great variety of seemingly unrelated devices, processes and phenomena. By using it, one can introduce order where there seems to be none, and discover general laws governing all these systems, starting from the most general assumptions. This is the object of the theory of finite automata.

3.4. SEQUENTIAL MACHINES

Consider a system (Fig. 3.3) consisting of (a) a finite automaton A, which converts symbols ρ of the alphabet $\{\rho\}$ into symbols $\varkappa$ of the alphabet $\{\varkappa\}$ as per Eq. (3.5′) or (3.5″), where the function F is given, and (b) a converter Φ which instantaneously and uniquely matches each symbol $\varkappa$ with a symbol λ from an alphabet $\{\lambda\}$:

$$\lambda^p = \Phi(\varkappa^p). \tag{3.8}$$

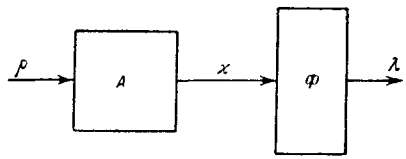


Fig. 3.3.

The instants at which the symbols ρ and λ appear coincide with the discrete time scale specific to the given automaton A. If one can select a single-valued function F^* (which may differ from F) in such a way as to make symbols λ satisfy

a relation of the form (3.5′) or (3.5″), that is, if

$$\lambda^{p+1} = F^*[\lambda^p, \rho^p] \quad \text{or} \quad \lambda^{p+1} = F^*[\lambda^p, \rho^{p+1}], \tag{3.9}$$

then the system consisting of automaton A and converter Φ will also be a finite automaton. Naturally, such a function F^* is not always available if for no other reason than that the alphabet $\{\lambda\}$ may differ from the alphabet $\{\varkappa\}$ in the number of symbols it contains; that is, the same symbol λ may be used to code several symbols $\varkappa$.

For example, let the alphabets $\{\varkappa\}$ and $\{\lambda\}$ consist of eight and two symbols, respectively, and let the converter Φ generate the symbol λ_1 in response to an input of symbols $\varkappa_1$ to $\varkappa_4$ or the symbol λ_2 when the input consists of any of the symbols $\varkappa_5$ to $\varkappa_8$. We shall assume that Eq. (3.5) holds and that F is such that after $\rho^p = \rho_1$, $\varkappa^p = \varkappa_1$ there follows the symbol $\varkappa^{p+1} = \varkappa_3$, and after $\rho^p = \rho_1$, $\varkappa^p = \varkappa_4$ there follows the symbol $\varkappa^{p+1} = \varkappa_6$. In the first instance the counting device will register

$$\rho^p = \rho_1, \quad \lambda^p = \lambda_1, \quad \lambda^{p+1} = \lambda_1,$$

and in the second case

$$\rho^p = \rho_1, \quad \lambda^p = \lambda_1, \quad \lambda^{p+1} = \lambda_2.$$

Thus, identical λ^p and ρ^p may be followed by different λ^{p+1}. This means that our automaton-converter system is not in itself an automaton, since it does not preserve a relation of the form (3.5′) between symbols λ and ρ.

The system shown in Fig. 3.3 is, however, a finite dynamical system. We shall call it a *finite automaton with an output converter*, or simply a *finite automaton with output*. In this case, the symbols λ are called the *output symbols* (as distinct from $\varkappa$, the *state symbols)*, the alphabet $\{\lambda\}$ is called the *output alphabet*, and the converter Φ is called the *output converter*.

In a more general case, the converter may have two inputs. The symbols fed to one of them are again from the alphabet $\{\varkappa\}$, while the signals to the other are symbols ρ. The converter then instantaneously matches a symbol λ with each $(\varkappa, \rho)$ pair (Fig. 3.4). A finite dynamical system obtained by coupling a finite automaton to an output converter which admits the symbol ρ (Fig. 3.4) is called a *sequential machine* (or, briefly, an *s-machine*). Of course, an s-machine may also be a finite automaton. Whether it is or not depends on the form of the function F in Eqs. (3.5) for the automaton

A, as well as the type of converter Φ used. In all cases, however, the system of Fig. 3.4 is a finite dynamical system.

A sequential machine becomes a finite automaton (that is, operates as a finite automaton) if the values of λ (the output) are uniquely defined by the value λ at the preceding discrete moment and the value of ρ at the current moment, that is, if the relationship

$$\lambda^{p+1} = F^*[\lambda^p, \rho^{p+1}]$$

holds. This, for example, will be the case when one uses an identity converter whose alphabet $\{\lambda\}$ coincides with the alphabet $\{\varkappa\}$, that is, a converter that generates a symbol λ^p coinciding with the input symbol $\varkappa^p$ regardless of ρ^p. In this sense, the concept of a "finite automaton" is a special case of the abstraction "sequential machine."

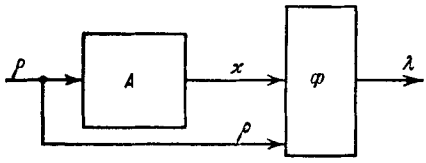


Fig. 3.4.

A finite automaton with an output converter may be treated as a special case of an s-machine in which the function Φ is independent of ρ.

At a first glance, the concept of a sequential machine appears broader than that of a finite automaton with output. However, this is not the case. This will be proven in Section 4.3, after we have formulated the concept of a "net."

A sequential machine is of the P - P or P - Pr type, depending on the automaton it contains. We shall now consider an arbitrary s-machine of the P - Pr type:

$$\varkappa^{p+1} = F[\varkappa^p, \rho^{p+1}], \quad \lambda^p = \Phi[\varkappa^p, \rho^p]. \tag{3.10}$$

Eliminating the symbol $\varkappa^p$ from the converter equation, we get

$$\lambda^{p+1} = \Phi[F(\varkappa^p, \rho^{p+1})\rho^{p+1}].$$

Let us now introduce the symbol $\tilde{\varkappa}$, which is defined in the alphabet

$\{\varkappa\}$ by the relation

$$\tilde{\varkappa}^{p+1} = \varkappa^p.$$

After substitution, we obtain

$$\lambda^p = \Phi[F(\tilde{\varkappa}^p, \rho^p), \rho^p] = \Phi^*(\tilde{\varkappa}^p, \rho^p).$$

If we now employ $\tilde{\varkappa}$ in Eq. (3.10) for the automaton, we obtain

$$\tilde{\varkappa}^{p+1} = F[\tilde{\varkappa}^p, \rho^p];$$

that is, all these transformations give an s-machine of the P - P type:*

$$\tilde{\varkappa}^{p+1} = F(\tilde{\varkappa}^p, \rho^p), \quad \lambda^p = \Phi^*(\tilde{\varkappa}^p, \rho^p). \tag{3.11}$$

Thus, any P- Pr type s-machine may be transformed into P - P type s-machine merely by replacing its output converter Φ by a Φ^* converter. However, the reverse is not generally true. We shall return to this problem in Section 5.4.

3.5. TECHNIQUES FOR DEFINING FINITE AUTOMATA AND SEQUENTIAL MACHINES

Any function

$$z = F(x, y),$$

where x and y assume values from finite sets, may be given by a table such as 3.1 showing the corresponding values of z.

The equation of finite automaton of the P - P type

$$\varkappa^{p+1} = F[\varkappa^p, \rho^p]$$

or the P - Pr type

$$\varkappa^{p+1} = F[\varkappa^p, \rho^{p+1}]$$

may be represented by an analogous Table 3.2, in which the $\varkappa^{p+1}$ symbol for a P - P automaton is represented by the intersection of row $\varkappa^p$ and column ρ^p, while the intersection of row $\varkappa^p$ and column ρ^{p+1} defines the symbol $\varkappa^{p+1}$ for a P - Pr automaton. We shall call this the *basic table* of the finite automaton.

*A finite automaton with an output converter described by Eqs. (3.5') and (3.8) is frequently referred to as a Moore machine (see [73]), and an s-machine given by Eq. (3.11) is called a Mealy machine (see [190]).

Table 3.1

x \ y	y_1	y_2	...	y_m
x_1	z_2	z_1	...	z_6
x_2	z_5	z_1	...	z_2
...	...	...	...	...
x_n	z_4	z_5	...	z_3

Table 3.2

x^p \ p	p_1	p_2	...	p_r
x_1	x_3	x_2	...	x_1
x_2	x_1	x_6	...	x_k
...	...	...	...	...
x_k	x_5	x_4	...	x_5

Each column of such a basic table is, in turn, the basic table for an autonomous automaton, which is obtained from the finite one by making the p value in the heading of that column a constant.

Consider one such autonomous automaton (for example, Table 3.3). If we draw k circles on a plane, assign to each circle a symbol x, and draw arrows which show the transitions occurring in the autonomous automaton in accordance with its basic table, we obtain its *graph*. Only one arrow can start at each circle, but any number of arrows (not exceeding k, however) may terminate at it. The graph of Fig. 3.5 corresponds to Table 3.3. Since each finite automaton yields r autonomous ones, the basic table of one finite automaton yields r graphs of autonomous automata. Figure 3.6 shows the graphs derived from Table 3.4.

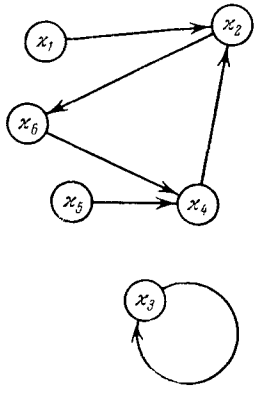


Fig. 3.5.

Since all these r graphs consist of the same k circles, they can be combined into one graph, in which each arrow is labeled with the p value at which that arrow can be performed. These labels are placed at the origin of the arrow in the case of the P - P automaton, and at its tip in the case of the P - Pr machine. Figure 3.7 shows the combined graph for Table 3.4 and Fig. 3.6, that is, for a P - Pr automaton.

Such a graph is called the *state diagram* of the automaton. In this case, each circle is the origin of r arrows. If several of these terminate at the same circle, they may be combined into one, the labels indicating their p values being joined by means of disjunction signs. The *state diagram* is fully equivalent and interconvertible with the basic table.

Table 3.3

$\varkappa$ \ ρ	$\rho_j = \rho^*$
$\varkappa_1$	$\varkappa_2$
$\varkappa_2$	$\varkappa_6$
$\varkappa_3$	$\varkappa_3$
$\varkappa_4$	$\varkappa_2$
$\varkappa_5$	$\varkappa_4$
$\varkappa_6$	$\varkappa_4$

Table 3.4

$\varkappa$ \ ρ	ρ_1	ρ_2	ρ_3
$\varkappa_1$	$\varkappa_5$	$\varkappa_6$	$\varkappa_6$
$\varkappa_2$	$\varkappa_4$	$\varkappa_4$	$\varkappa_4$
$\varkappa_3$	$\varkappa_3$	$\varkappa_3$	$\varkappa_2$
$\varkappa_4$	$\varkappa_3$	$\varkappa_2$	$\varkappa_5$
$\varkappa_5$	$\varkappa_6$	$\varkappa_5$	$\varkappa_1$
$\varkappa_6$	$\varkappa_2$	$\varkappa_5$	$\varkappa_3$

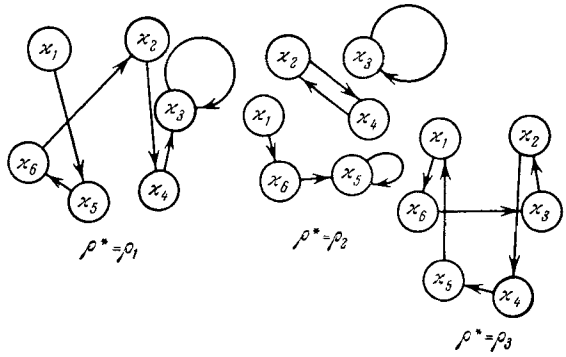


Fig. 3.6.

Let us construct a square $k \times k$ matrix C whose rows (from top to bottom) and columns (from left to right) are headed by symbols $\varkappa_1, \varkappa_2, \ldots, \varkappa_k$. The matrix element at the intersection of the $\varkappa_q$-th row and the $\varkappa_s$-th column is the label of the state diagram arrow that connects the circle $\varkappa_q$ with the circle $\varkappa_s$. This element may consist of one symbol ρ or a disjunction of several ρ's. If there is no arrow between a circle $\varkappa_l$ and a circle $\varkappa_m$, then the corresponding square of matrix C contains a 0. Thus we obtain the matrix of Table 3.5 for the state diagram of Fig. 3.7.

This matrix is called an ***interconnection*** (or transition) ***matrix*** and is still another way of defining a finite automaton. Here, just as in the case of a basic table, one must specify beforehand whether the

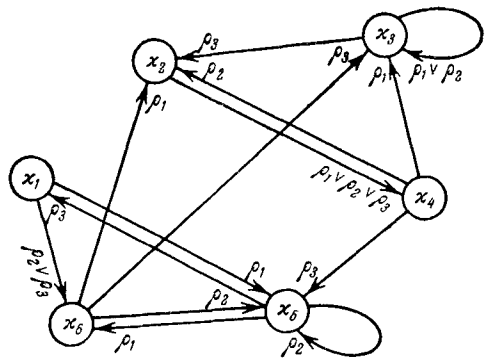


Fig. 3.7.

Table 3.5

	x_1	x_2	x_3	x_4	x_5	x_6
x_1	0	0	0	0	ρ_1	$\rho_2 \vee \rho_3$
x_2	0	0	0	$\rho_1 \vee \rho_2 \vee \rho_3$	0	0
x_3	0	ρ_3	$\rho_1 \vee \rho_2$	0	0	0
x_4	0	ρ_2	ρ_1	0	ρ_3	0
x_5	ρ_3	0	0	0	ρ_2	ρ_1
x_6	0	ρ_1	ρ_3	0	ρ_2	0

automaton is of the P - P or the P - Pr type. Each row of the matrix must contain every ρ_i once, and only once. The matrix may be derived directly from the basic table, dispensing with the intermediate state diagram.

Assume we have a basic table for a P - P automaton, that is, for the relation

$$x^{p+1} = F[x^p, \rho^p].$$

Then each cell of the basic table defines three symbols:

$$x^p, \rho^p, x^{p+1},$$

that is, the row heading, the column heading, and the character contained in the cell. Let us call such a symbol triplet a triad. Since the whole table has rk squares, and each square defines a triad, the table defines a set of rk triads. We shall say that a set of triads is ordered if the first two symbols (x^p and ρ^p) of any two triads of that

set do not coincide. Any basic table of a finite automaton defines an ordered finite set of triads. Conversely, any ordered set of rk triads defines a basic table, that is, a finite automaton.

The above also applies to P - Pr automata, that is, those defined by

$$\varkappa^{p+1} = F[\varkappa^p, \rho^{p+1}].$$

However, in this case each triad consists of the symbol triplet

$$\varkappa^p, \rho^{p+1}, \varkappa^{p+1}$$

so that an ordered set is said to be one in which no two triads have identical first two symbols $\varkappa^p$ and ρ^{p+1}.

Now, if we wish to define a sequential machine in a table form, we must consider simultaneously the equations of its constituent automaton [(3.5′) or (3.5″)] and converter (3.8). To do this, we draw up the basic table for the finite automaton and add to each square the symbol λ^{p+1} resulting from the converter equation. This combined table is the *basic table of the s-machine*.

If the automaton is of the P - P type [Eq. (3.5′)], then we add to its table symbol λ^p. For example, the automaton of Table 3.4 plus the converter of Table 3.6 define the s-machine basic table 3.7. If, however, the automaton is of the P - Pr type [Eq. (3.5″)], then we add to each square the symbol λ^{p+1}, obtained from the converter table at the intersection of $\varkappa^{p+1}$ (the symbol already present in the square of the basic table) and ρ^{p+1} (the heading of the basic table column in which the square is situated). Thus, if the P - Pr automaton is given by Table 3.4 and the associated converter by Table 3.6, then the basic table of the s-machine incorporating this P - Pr automaton is Table 3.8.

Let us note in passing that the converter table for the P - Pr case may contain blank spaces, because some $\varkappa$ values may be missing from the corresponding column of the automaton table. Thus we could leave the square $(\rho_1, \varkappa_1)$ of Table 3.6 blank, because column ρ_1 of Table 3.4. contains no $\varkappa_1$ entries.

To obtain a state diagram of an s-machine, we modify the diagram of the corresponding automaton by including the appropriate λ symbol at each circle. However, in an s-machine λ is defined not only by $\varkappa$, but also by ρ. For this reason the state diagram for an s-machine differs from that of a finite automaton with an associated output converter in that the symbol λ is not written inside a circle, but *side by side with the symbol* ρ *(above the arrow)*. The arrow connects state $\varkappa^p$ with state $\varkappa^{p+1}$. For a P - P automaton [Eq. (3.5′)],

Table 3.6

x \ ρ	ρ_1	ρ_2	ρ_3
x_1	λ_1	λ_2	λ_3
x_2	λ_2	λ_2	λ_3
x_3	λ_1	λ_1	λ_3
x_4	λ_3	λ_1	λ_2
x_5	λ_1	λ_1	λ_2
x_6	λ_3	λ_3	λ_1

Table 3.7

x^p \ ρ^p	ρ_1	ρ_2	ρ_3
x_1	x_5, λ_1	x_6, λ_2	x_6, λ_3
x_2	x_4, λ_2	x_4, λ_2	x_4, λ_3
x_3	x_3, λ_1	x_3, λ_1	x_2, λ_3
x_4	x_3, λ_3	x_2, λ_1	x_5, λ_2
x_5	x_6, λ_1	x_5, λ_1	x_1, λ_2
x_6	x_2, λ_3	x_5, λ_3	x_3, λ_1

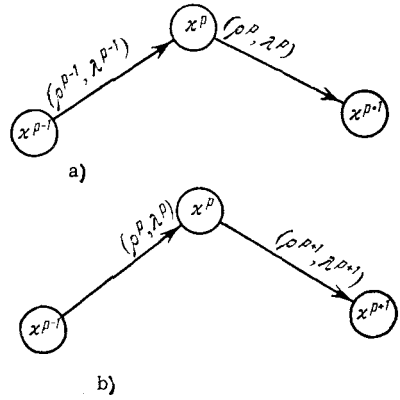


Fig. 3.8.

the symbol pair (ρ, λ) is written at the origin of the arrow (Fig. 3.8,a), whereas in the case of P - Pr automaton [Eq. (3.5'')], the label is at its tip (Fig. 3.8,b). In the first case, the output λ is defined in the converter table at the intersection of the symbol ρ, written above the arrow, and the symbol x, written inside the circle from which the arrow emerges; in the second case, the coordinates are the ρ above the arrow and the x in the circle at the tip of the arrow. Thus, Fig. 3.9 shows a state diagram based on Table 3.8.

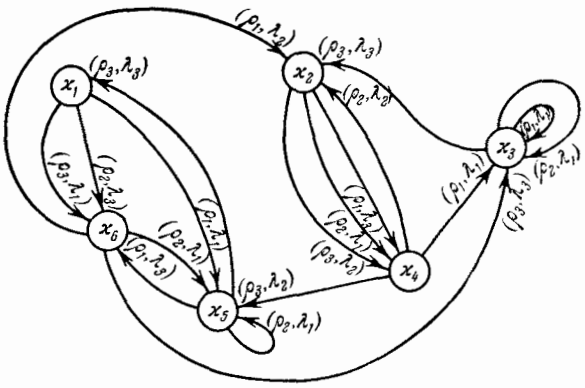


Fig. 3.9.

The state diagram of a sequential machine again gives an interconnection matrix. This matrix differs from that describing a finite automaton in that its elements consist of labels at the tips of the arrows of the state diagram, that is, the symbol pairs (ρ, λ). If the basic table is 3.8, then the interconnection matrix is that of Table 3.9.

Table 3.8

x^p \ ρ^{p+1}	ρ_1	ρ_2	ρ_3
x_1	x_5, λ_1	x_6, λ_3	x_6, λ_1
x_2	x_4, λ_3	x_4, λ_1	x_4, λ_2
x_3	x_3, λ_1	x_3, λ_1	x_2, λ_3
x_4	x_3, λ_1	x_2, λ_2	x_5, λ_2
x_5	x_6, λ_3	x_5, λ_1	x_1, λ_3
x_6	x_2, λ_2	x_6, λ_1	x_3, λ_3

3.6. RECORDING THE OPERATION OF AN AUTOMATON

So far, we have established that symbols ρ are sequentially "communicated" to the automaton (or s-machine) from the outside, and that they are independent of its operation. The machine then processes the input ρ into symbols x (or λ).

Now we shall call an *input sequence* any finite (but as large as desired) series of ρ symbols, and we shall call the analogous set of x or λ symbols a *state sequence* (or an *output sequence*). The number of symbols contained in such a set will be called the *length of the sequence*.

Table 3.9

	x_1	x_2	x_3	x_4	x_5	x_6
x_1	0	0	0	0	(ρ_1, λ_1)	$(\rho_2, \lambda_3) \vee (\rho_3, \lambda_1)$
x_2	0	0	0	$(\rho_1, \lambda_3) \vee (\rho_2, \lambda_1) \vee (\rho_3, \lambda_2)$	0	0
x_3	0	(ρ_3, λ_3)	$(\rho_2, \lambda_1) \vee (\rho_1, \lambda_1)$	0	0	0
x_4	0	(ρ_2, λ_2)	(ρ_1, λ_1)	0	(ρ_3, λ_2)	0
x_5	(ρ_3, λ_3)	0	0	0	(ρ_2, λ_1)	(ρ_1, λ_3)
x_6	0	(ρ_1, λ_2)	(ρ_3, λ_3)	0	(ρ_2, λ_1)	0

Both the automaton and the sequential machine are operators that process sequences of input symbols of one alphabet into sequences of output symbols of another alphabet. The basic table, the set of triads, the graphs, the state diagram, the interconnection matrix, and the transition table are various methods of defining such an operator, since any one of these is sufficient to recreate the corresponding sequence $\varkappa$ (or λ) if the sequence ρ and the initial state $\varkappa^0$ are known.

Table 3.10

Discrete moment	0	1	2	3	4	5	6	7	8	...
ρ	ρ_3	ρ_8	ρ_1	ρ_3	ρ_4	ρ_6	ρ_2	ρ_8	ρ_{12}	...
$\varkappa$	$\varkappa_1$	$\varkappa_2$	$\varkappa_7$	$\varkappa_1$	$\varkappa_2$	$\varkappa_{12}$	$\varkappa_6$	$\varkappa_3$	$\varkappa_8$	...

Consider now a three-row table 3.10, whose row 1 contains the ordinal number of the discrete time moment, and rows 2 and 3 contain the corresponding ρ and $\varkappa$. The table may have as many columns as desired. This is the *tape* of the finite automaton. Since a tape may be compiled for each input sequence $\rho(t)$, each automaton may have an infinite number of tapes.

Any initial piece of tape (from zero to any kth moment) contains an input sequence and the corresponding state sequence of length $k + 1$. Any three tape symbols, such as those delineated by the heavy line in Table 3.10, constitute a triad defining one cell of the basic table of the automaton (we shall use a heavy line to isolate a P - P triad, and a dotted line to delineate a P - Pr triad). Now, if we had a scanner with a cutout matching either the heavy or the dotted outline of the Table 3.10, then, sliding this scanner along the tape, we would consecutively see all the various triads contained in it (obviously, each tape has a finite number of such triads). If we could scan all the tapes of a given automaton (an infinite number), we could then read the set of all the triads contained in all the tapes. But since all these tapes are generated by a single automaton, the set of triads must be finite. In fact, it is the finite ordered set of triads containing rk elements.

In the preceding section we showed that since an automaton is an operator, it can be defined by a finite, ordered set of triads. We see now that all the tapes of the automaton consist of triads of this set.

Let us now select an alphabet of rk symbols, for instance, $\{\tau\}$, and assign symbols τ to all the triads of our automaton. Then the tape

shall be reduced to only two rows: the ordinal number of the discrete moment and, the symbol τ (see Table 3.11). However, this reduces the available degrees of freedom, for the sequence of triads in such a tape cannot be arbitrary. Indeed, let the first triad τ_5 consist of

$$\varkappa_7,\ \rho_3,\ \varkappa_{12}.$$

This immediately fixes the first symbol in the following triad, so that only the two remaining symbols can vary; that is, the next triad can only be some triad of the same set which starts with $\varkappa_{12}$, for instance, $\varkappa_{12}, \rho_8, \varkappa_6$ or $\varkappa_{12}, \rho_2, \varkappa_{12}$, and so on. The triads corresponding to a given triad τ_j are those triads from our ordered set whose first symbol coincides with the last symbol of the given triad τ_j. All the tapes of a finite automaton consist of triads arranged in such a way that each triad is followed by any one of its corresponding triads.

Table 3.11

Discrete moment	0	1	2	3	4	5	6	...
τ	τ_5	τ_1	τ_3	τ_2	τ_3	τ_1	τ_8	...

The concept of a tape may be extended to the sequential machine by supplementing the finite automaton tape with a row of output symbols λ, after which the $\varkappa$ row is deleted (Table 3.12). This tape may also be split into triads such as

$$\lambda^p,\ \rho^p,\ \lambda^{p+1} \text{ or } \lambda^p,\ \rho^{p+1},\ \lambda^{p+1}.$$

This set of triads may contain, however, some ''contradictory'' elements, in which the first two symbols λ^p, ρ^p (or λ^p, ρ^{p+1}) are identical but are followed by differing third symbols λ^{p+1}, that is, the set is not an ordered one. It becomes ordered if, and only if, the sequential machine as a whole is a finite automaton.

Table 3.12

Discrete moment	0	1	2	3	4	5	6	7	8	...
ρ	ρ_3	ρ_8	ρ_1	ρ_3	ρ_9	ρ_6	ρ_7	ρ_8	ρ_{12}	...
λ	λ_2	λ_3	λ_3	λ_2	λ_1	λ_1	λ_1	λ_2	λ_3	...

We shall now discuss still another way of describing the operation of a finite automaton. Let us draw its state diagram (Fig. 3.10)

and consider the two circles incorporating symbols $\varkappa_i$ and $\varkappa_j$, respectively. The transition from state $\varkappa_i$ to state $\varkappa_j$ may be accomplished over one discrete moment, provided the input is ρ_1. The transition over two moments may be accomplished via the following alternative routes: in the first moment, the input is ρ_2 or ρ_3, and in the second moment it is ρ_2; or it is ρ_1 in both moments. If three discrete moments are available, then one can accomplish the transition via nine different alternative routes (Table 3.13). Similarly, we can derive all possible sequences of ρ that would transform the state $\varkappa_i$ into the state $\varkappa_j$ over q discrete moments. Each such sequence is a *path of length* q leading from $\varkappa_i$ to $\varkappa_j$, and we shall represent it as a sequence of q symbols ρ; the aggregate of all the possible paths of length q shall be represented as a disjunction of such sequences. Thus, for example, Table 3.13 may be written in the form

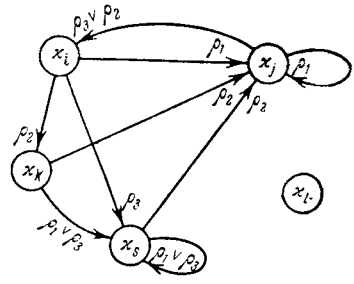


Fig. 3.10.

$$\rho_1\rho_1\rho_1 \vee \rho_1\rho_3\rho_1 \vee \rho_2\rho_1\rho_2 \vee \rho_2\rho_3\rho_2 \vee \rho_3\rho_1\rho_2 \vee \rho_3\rho_3\rho_2 \vee \rho_1\rho_2\rho_1 \vee \rho_2\rho_2\rho_1 \vee \rho_3\rho_2\rho_1 .$$

Let us denote by D^q_{ij} a disjunction describing all the possible $\varkappa_i$ to $\varkappa_j$ paths of length q. Then each disjunction $\{\rho\}$, shall comprise one or more sequences consisting of exactly q elements ρ_i, some of which may coincide.

Table 3.13

Path	Discrete moment		
	0	1	2
First	ρ_1	ρ_1	ρ_1
Second	ρ_1	ρ_3	ρ_1
Third	ρ_2	ρ_1	ρ_2
Fourth . . .	ρ_2	ρ_3	ρ_2
Fifth	ρ_3	ρ_1	ρ_2
Sixth	ρ_3	ρ_3	ρ_2
Seventh . . .	ρ_1	ρ_2	ρ_1
Eighth	ρ_2	ρ_2	ρ_1
Ninth	ρ_3	ρ_2	ρ_1

Let us now construct a matrix $C^{(q)}$. This matrix will contain D^q_{ij} (at the intersection of the ith row and the jth column) if there is at least one $\varkappa_i$ to $\varkappa_j$ path of length q, and 0 if there is no such path. For example,

$$C^{(q)} = \begin{array}{c} \\ \varkappa_1 \\ \varkappa_2 \\ \varkappa_3 \\ \varkappa_4 \end{array} \begin{array}{c} \begin{array}{cccc} \varkappa_1 & \varkappa_2 & \varkappa_3 & \varkappa_4 \end{array} \\ \begin{bmatrix} D^{(q)}_{11} & 0 & 0 & D^{(q)}_{14} \\ 0 & 0 & D^{(q)}_{23} & 0 \\ 0 & D^{(q)}_{32} & D^{(q)}_{33} & D^{(q)}_{34} \\ 0 & D^{(q)}_{42} & 0 & 0 \end{bmatrix} \end{array}.$$

This is the *matrix of the path of length* q. Just as in the interconnection matrix, the rows (top to bottom) and the columns (left to right) correspond to symbols $\varkappa_1, \varkappa_2, \ldots, \varkappa_k$ and are so denoted.

The matrix $C^{(q)}$ contains all the paths leading from any initial state to any final state (which may coincide with the initial state) over q discrete moments. Because any sequence consisting of q symbols ρ will transform the automaton from any state into the same or another state, each row of the matrix $C^{(q)}$ contains, but only once, all the possible sequences that may be formed from the alphabet $\{\rho\}$ by selecting q symbols at a time. Thus, for example, each row of the matrix $C^{(2)}$ must contain the sequences

$$\rho_1\rho_1;\ \rho_2\rho_2;\ \rho_3\rho_3;\ \rho_1\rho_2;\ \rho_2\rho_1;\ \rho_1\rho_3;\ \rho_3\rho_1;\ \rho_3\rho_2;\ \rho_2\rho_3.$$

Each row of the $C^{(q)}$ matrix may contain these groups of q symbols in different disjunctive arrangements and they may be distributed over different columns, in accordance with the basic table for a given automaton.

A complete set of matrices $C^{(1)}, C^{(2)}, C^{(3)}, \ldots$ completely specifies the operation of the automaton over any desired time period. However, such a set is not very useful and not really needed because we can always rederive the entire set of matrices from the starting interconnection matrix. We do this as follows.

To begin with, $C^{(1)} = C$, that is, the matrix of path length 1 coincides with the interconnection matrix since, by definition, its elements are those values ρ which transform $\varkappa_i$ to $\varkappa_j$ over one discrete moment.

Let us now square the interconnection matrix, in accordance with the following rules.

1. An element $C^2{}_{ij}$ of the product matrix (located at the intersection of the ith row and jth column) is specified, in accordance with general rules of matrix multiplication, as the sum of the

products of the elements of the ith row of the first factor by the elements of the jth column of the second factor. These products are not commutative; that is, in multiplying the elements, the positions of the factors cannot be interchanged.

2. Addition signs are replaced by disjunction signs throughout.
3. Multiplication signs define the operation of assigning symbols ρ.

For example, consider the matrix C for the state diagram of Fig. 3.7:

$$C = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{array} \begin{array}{c} \begin{array}{cccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & \rho_1 & \boxed{\rho_2 \vee \rho_3} \\ 0 & 0 & 0 & \rho_1 \vee \rho_2 \vee \rho_3 & 0 & 0 \\ 0 & \rho_3 & \rho_1 \vee \rho_2 & 0 & 0 & 0 \\ 0 & \rho_2 & \rho_1 & 0 & \rho_3 & 0 \\ \rho_3 & 0 & 0 & 0 & \rho_2 & \rho_1 \\ 0 & \boxed{\rho_1} & \rho_3 & 0 & \rho_2 & 0 \end{bmatrix} \end{array}$$

and square it in accordance with the above rules:

$$C^2 = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{array} \begin{array}{c} \begin{array}{cccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{array} \\ \begin{bmatrix} \rho_1\rho_3 & \boxed{\rho_2\rho_1 \vee \rho_3\rho_1} & \rho_2\rho_3 \vee \rho_3\rho_3 & 0 & \rho_1\rho_2 \vee \rho_2\rho_2 \vee \vee \rho_3\rho_2 & \rho_1\rho_1 \\ 0 & \rho_1\rho_2 \vee \rho_2\rho_2 \vee \vee \rho_3\rho_2 & \rho_1\rho_1 \vee \rho_2\rho_1 \vee \vee \rho_3\rho_1 & 0 & \rho_1\rho_3 \vee \rho_2\rho_3 \vee \vee \rho_3\rho_3 & 0 \\ 0 & \rho_1\rho_3 \vee \rho_2\rho_3 & \rho_1\rho_1 \vee \rho_2\rho_1 \vee \vee \rho_1\rho_2 \vee \rho_2\rho_2 & \rho_3\rho_1 \vee \rho_3\rho_2 \vee \vee \rho_3\rho_3 & 0 & 0 \\ \rho_3\rho_3 & \rho_1\rho_3 & \rho_1\rho_1 \vee \rho_1\rho_2 & \rho_2\rho_1 \vee \rho_2\rho_2 \vee \vee \rho_2\rho_3 & \rho_3\rho_2 & \rho_3\rho_1 \\ \rho_2\rho_3 & \rho_1\rho_1 & \rho_1\rho_3 & 0 & \rho_3\rho_1 \vee \rho_2\rho_2 \vee \vee \rho_1\rho_2 & \rho_3\rho_2 \vee \rho_3\rho_3 \vee \vee \rho_2\rho_1 \\ \rho_2\rho_3 & \rho_3\rho_3 & \rho_3\rho_1 \vee \rho_3\rho_2 & \rho_1\rho_1 \vee \rho_1\rho_2 \vee \vee \rho_1\rho_3 & \rho_2\rho_2 & \rho_2\rho_1 \end{bmatrix} \end{array}.$$

We see that $C^2 = C^{(2)}$, that is, matrix C^2 is composed of all the possible paths of Fig. 3.7 which lead from one circle to another over two discrete moments. For example, element $C_{12}{}^2$ consists of $\rho_2\rho_1 \vee \rho_3\rho_1$; thus, paths $\rho_2\rho_1$ and $\rho_3\rho_1$ are the only paths over which x_1 can be transformed into x_2 in two moments.

The coincidence of C^2 and $C^{(2)}$ in this case is no accident. We see that $C_{12}{}^2$ is the product of nonzero elements C_{16} and C_{62}. However, such nonzero elements in C indicate that each of the transformations x_1 to x_6 and x_6 to x_2 requires one discrete moment, that is, that there

exist $\varkappa_1$ to $\varkappa_2$ paths which can be traversed in two moments. Therefore the coincidence found in this example is actually a general rule which states that *the square of the interconnection matrix is the matrix of all the two-moment paths*: $C^2 = C^{(2)}$. By analogy $C^3 = C^{(3)}$, that is, the cube of the interconnection matrix is the matrix of all the three-moment paths and, generally, *a matrix of all the q-moment paths is obtained by raising the interconnection matrix to the power of q*:

$$C^{(q)} = C^q.$$

We now see why we do not need the set $C^{(1)}, C^{(2)}, C^{(3)}\ldots$: all the possible paths over any number of moments q can be derived via multiplication of the interconnection matrix by itself.

All of the above also pertains to sequential machines. One must only remember that each arrow of the state diagram for the s-machine carries the two symbols ρ and λ, so that a transition in the s-machine is characterized by a pair (ρ_k, λ_s).

The "operation of symbol assignment" is then performed in accordance with the following rules:

$$1.\ (\rho_i, \lambda_j)(\rho_k, \lambda_s) = (\rho_i\rho_k, \lambda_j\lambda_s),$$
$$2.\ (\rho_k, \lambda_s)0 = 0(\rho_k, \lambda_s) = 0.$$

If one wants to multiply disjunctions of pairs, one utilizes the distributive property of the operation of multiplication of disjunctions. For example,

$$[(\rho_1, \lambda_3) \vee (\rho_2, \lambda_5)][(\rho_8, \lambda_1) \vee (\rho_5, \lambda_3)] =$$
$$= (\rho_1\rho_8, \lambda_3\lambda_1) \vee (\rho_2\rho_8, \lambda_5\lambda_1) \vee (\rho_1\rho_5, \lambda_3\lambda_3) \vee (\rho_2\rho_5, \lambda_5\lambda_3).$$

As an example, consider the interconnection matrix of the tri-state s-machine whose state diagram is shown in Fig. 3.11.

$$C = \begin{array}{c} \\ \varkappa_1 \\ \varkappa_2 \\ \varkappa_3 \end{array} \begin{array}{c} \begin{array}{ccc} \varkappa_1 & \varkappa_2 & \varkappa_3 \end{array} \\ \begin{bmatrix} 0 & (\rho_1, \lambda_3) \vee (\rho_2, \lambda_1) & 0 \\ 0 & (\rho_2, \lambda_2) & (\rho_1, \lambda_1) \\ (\rho_2, \lambda_3) & (\rho_1, \lambda_2) & 0 \end{bmatrix} \end{array}.$$

On squaring this matrix, we obtain

$$C^2 = \begin{array}{c} \\ \varkappa_1 \\ \varkappa_2 \\ \varkappa_3 \end{array} \begin{array}{c} \begin{array}{ccc} \varkappa_1 & \varkappa_2 & \varkappa_3 \end{array} \\ \begin{bmatrix} 0 & (\rho_1\rho_2, \lambda_3\lambda_2) \vee (\rho_2\rho_2, \lambda_1\lambda_2) & (\rho_1\rho_1, \lambda_3\lambda_1) \vee \\ & & \vee (\rho_2\rho_1, \lambda_1\lambda_1) \\ (\rho_1\rho_2, \lambda_1\lambda_3) & (\rho_2\rho_2, \lambda_2\lambda_2) \vee (\rho_1\rho_1, \lambda_1\lambda_2) & (\rho_2\rho_1, \lambda_2\lambda_1) \\ 0 & (\rho_2\rho_1, \lambda_3\lambda_3) \vee (\rho_2\rho_2, \lambda_3\lambda_1) \vee & (\rho_1\rho_1, \lambda_2\lambda_1) \\ & \vee (\rho_1\rho_2, \lambda_2\lambda_2) & \end{bmatrix} \end{array}.$$

The very construction of matrix C^2 shows that the element C^2_{ij} represents the list of all the input (and the corresponding output) sequences of length 2 that will transform the ith state of the s-machine into its jth state over two discrete moments. Thus, in the case of the s-machine shown in Fig. 3.11, sequences $\rho_2\rho_1$, $\rho_2\rho_2$ and $\rho_1\rho_2$ transform the third state into the second, a transformation accompanied by the appearance of sequences $\lambda_3\lambda_3$, $\lambda_3\lambda_1$ or $\lambda_2\lambda_2$ at the output.

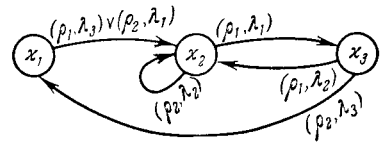


Fig. 3.11.

The matrix C^2 can again be multiplied by C, using analogous rules. The only difference is that the multiplication now involves elements that may contain not only symbol pairs (ρ_s, λ_t), but also pairs of symbol sequences $(\rho_i\rho_j, \lambda_k\lambda_l)$. Multiplication then means the operation of assigning symbols, for example:

$$(\rho_3\rho_5, \lambda_6\lambda_7)(\rho_4, \lambda_2) = (\rho_3\rho_5\rho_4, \lambda_6\lambda_7\lambda_2).$$

In multiplying elements containing disjunction one also uses the distributive law.

In our example the matrix C^3 has the form

$$C^3 = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \end{array} \begin{array}{c} \begin{array}{ccc} x_1 & x_2 & x_3 \end{array} \\ \left[\begin{array}{ccc} [(\rho_1\rho_1\rho_2, \lambda_3\lambda_1\lambda_3) \vee & [(\rho_1\rho_2\rho_2, \lambda_3\lambda_2\lambda_2) \vee (\rho_2\rho_2\rho_2, \lambda_1\lambda_2\lambda_2) \vee & [(\rho_1\rho_2\rho_1, \lambda_3\lambda_2\lambda_1) \vee \\ \vee (\rho_2\rho_1\rho_2, \lambda_1\lambda_1\lambda_3)] & \vee (\rho_1\rho_1\rho_1, \lambda_3\lambda_1\lambda_2) \vee (\rho_2\rho_1\rho_1, \lambda_1\lambda_1\lambda_2)] & \vee (\rho_2\rho_2\rho_1, \lambda_1\lambda_2\lambda_1)] \\ (\rho_2\rho_1\rho_2, \lambda_2\lambda_1\lambda_3) & [(\rho_1\rho_2\rho_1, \lambda_1\lambda_3\lambda_3) \vee (\rho_1\rho_2\rho_2, \lambda_1\lambda_3\lambda_1) \vee & [(\rho_2\rho_2\rho_1, \lambda_2\lambda_2\lambda_1) \vee \\ & \vee (\rho_2\rho_2\rho_2, \lambda_2\lambda_2\lambda_2) \vee (\rho_1\rho_1\rho_2, \lambda_1\lambda_2\lambda_2) \vee & \vee (\rho_1\rho_1\rho_1, \lambda_1\lambda_2\lambda_1)] \\ & \vee (\rho_2\rho_1\rho_1, \lambda_2\lambda_1\lambda_2)] & \\ (\rho_1\rho_1\rho_2, \lambda_2\lambda_1\lambda_3) & [(\rho_2\rho_1\rho_2, \lambda_3\lambda_3\lambda_2) \vee (\rho_2\rho_2\rho_2, \lambda_3\lambda_1\lambda_2) \vee & [(\rho_2\rho_1\rho_1, \lambda_3\lambda_3\lambda_1) \vee \\ & \vee (\rho_1\rho_2\rho_2, \lambda_2\lambda_2\lambda_2) \vee (\rho_1\rho_1\rho_1, \lambda_2\lambda_1\lambda_2)] & \vee (\rho_2\rho_2\rho_1, \lambda_3\lambda_1\lambda_1) \vee \\ & & \vee (\rho_1\rho_2\rho_1, \lambda_2\lambda_2\lambda_1)] \end{array} \right] \end{array}.$$

The elements of the matrix C^3 thus indicate all the input sequences which can transform one state into another over three discrete moments, as well as the corresponding output sequences. Continuing the process of multiplying matrix C by itself, we finally obtain matrix C^q and thus find all the input sequences which transform state x_i into state x_j over q moments.*

*There are yet other methods of specifying the operation of an automaton. For example, Kobrinskiy and Trakhtenbrot [43] employ the "tree" of an automaton for this purpose, but we shall not use this concept.

3.7. ON THE RESTRICTION OF INPUT SEQUENCES

So far, we have assumed that the input symbol sequences may be random, provided that each of these symbols was contained in the alphabet $\{\rho\}$. Thus, if the alphabet $\{\rho\}$ consists of r symbols, we have at our disposal r^k different sequences of length k.

However, one frequently deals with problems in which one needs to examine only those symbol sequences which satisfy some special conditions. Sequences that satisfy such additional special restrictions are termed *admissible* (or allowable, or legitimate). For example, we could impose any of the following restrictions:

1. The only admissible sequences are those in which even subscripts of ρ alternate with odd ones. Under this restriction, the sequence $\rho_7\rho_2\rho_1\rho_4\rho_7\rho_6\rho_3 \ldots$ would be admissible, whereas the sequence $\rho_7\rho_2\rho_4\rho_1\rho_7 \ldots$ would not be.
2. The only admissible sequences are those in which no two identical symbols ρ are consecutive. In this case, the sequence $\rho_2\rho_7\rho_5\rho_3\rho_8\rho_1 \ldots$ would be admissible, whereas the sequence $\rho_2\rho_7\rho_5\rho_5\rho_3\rho_8\rho_8 \ldots$ would not be.
3. The only admissible sequences are those in which ρ_i is not immediately followed by ρ_j.

The restrictions imposed on sequences are often due to the manner in which the continuous time is divided into discrete intervals. Thus, if the next discrete moment occurs whenever the input is changed, then the restriction on the sequences $\rho(t)$ is that no two identical symbols may be consecutive. Similar restrictions always occur in the other cases in which the timing of the system is synchronized with some input "event."* Thus the restrictions on the input sequences can be of two kinds:

1. They may be imposed by some characteristic of timing of the system, in which case only admissible sequences will appear at the output.
2. They may have no connection with timing, that is, generally speaking, the s-machine can respond to any input sequences $\rho(t)$, but under the operating conditions, only admissible sequences do appear at its input. This distinction is immaterial to us at this point. There are, however, instances where the input sequences cannot be arbitrary but must satisfy some supplementary conditions. We shall discuss this subject later.

*We are relying on the reader's intuition in using the term "event" at this point, but we shall define it at a later stage.

In such cases the basic table alone is not sufficient for the definition of an automaton. Just as it must be supplemented with conditions defining the clock of the system, so in these instances it must be supplemented by the specification of a legitimate input sequence.

4

Abstract Structure and Nets

4.1. THE CONCEPT OF SUBSTITUTION OF SEQUENTIAL MACHINES

Consider two sequential machines: a machine s_1 (Fig. 4.1,a), which transforms symbols ρ of an alphabet $\{\rho\}$ into symbols $\varkappa$ of an alphabet $\{\varkappa\}$, and a machine s_2 (Fig. 4.1,b), which transforms symbols η of an alphabet $\{\eta\}$ into symbols ζ of an alphabet $\{\zeta\}$. Let us also

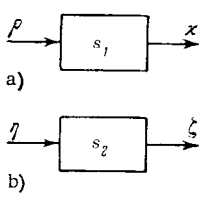


Fig. 4.1.

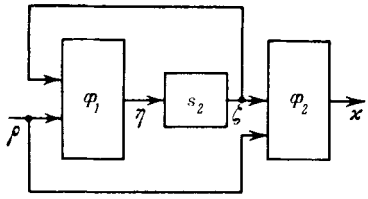


Fig. 4.2.

introduce the function converters Φ_1 and Φ_2, which perform

$$\eta = \Phi_1(\zeta, \rho),$$
$$\varkappa = \Phi_2(\zeta, \rho).$$

respectively. That is, the converter Φ_1 instantaneously and uniquely matches a symbol η of the alphabet $\{\eta\}$ with the symbol pair ζ and ρ from the alphabets $\{\zeta\}$ and $\{\rho\}$, respectively, whereas the converter Φ_2 matches a symbol $\varkappa$ of the alphabet $\{\varkappa\}$ with the above pair.

Let us couple the converters Φ_1 and Φ_2 with the sequential machine s_2 as illustrated in Fig. 4.2. The resulting system is a new s-machine that operates on the same alphabets as machine s_1.

If machines s_1 and s_2 are given, it may be possible to select converters Φ_1 and Φ_2 such that the resulting system of Fig. 4.2 will duplicate any result produced by the machine s_1. If that is possible,

we shall say that *the machine s_1 replaces the machine s_2* or, what is the same thing, that *the machine s_2 substitutes for the machine s_1*.

To give a strict definition of these terms, we must first define what we mean by the statement that a system "duplicates any result produced by a given s-machine." We shall agree that a machine s_2 substitutes for a machine s_1 if for each initial state $\varkappa^0$ of s_1 there exists at least one initial state ζ^0 of s_2 such that, for any input sequence of symbols from the alphabet $\{\rho\}$, both the system produced by coupling s_2 to appropriate converters Φ_1 and Φ_2 in the manner of Fig. 4.2 and the machine s_1 will generate the same output sequence of symbols from the alphabet $\{\varkappa\}$, starting from ζ^0 and $\varkappa^0$, respectively. The fact that the machine s_2 can be substituted for the machine s_1 will be indicated by:

$$s_2 \Rightarrow s_1.$$

When we write $s_2 \Rightarrow s_1$, we mean that the machine s_2, appropriately coupled with appropriate converters Φ_1 and Φ_2, can operate in the same way as the machine s_1, thus replacing it. In this sense, the system of Fig. 4.2 is also a sequential machine.

The fact that $s_2 \Rightarrow s_1$ does not necessarily mean that $s_1 \Rightarrow s_2$. Our definition of substitution is based on the complete independence of the choice of converters and initial states from the sequence of input symbols $\{\rho\}$. Naturally, we could have given a broader definition, relating the choice of converters and initial states to the input sequence. However, we are not concerned with such a broad concept (although it may be useful in some problems). We can also introduce the concept of *relative substitution* for an S-machine, if the set L of admissible input sequences of the machine s_1 is restricted.

The idea of substitution immediately involves the following problem: Given two s-machines s_1 and s_2 determine whether s_2 can substitute for s_1, that is, whether there exist function converters Φ_1 and Φ_2 such that the diagram of Fig. 4.2 describes a machine that substitutes for the machine s_1; if the answer is affirmative, construct converters Φ_1 and Φ_2. This problem has a trivial solution — all that is necessary is to test all the (finitely many) pairs of converters Φ_1 and Φ_2. If any of these pairs proves "suitable," then s_2 will be a substitute for s_1. Obviously, this search method is cumbersome and cannot be used in practice. However, the present authors know of no better method.

We shall now leave the generalized system of Fig. 4.2 and shall consider those of its special cases which are shown in Fig. 4.3. Of these, the system of Fig. 4.3,b is extremely important. Here, each

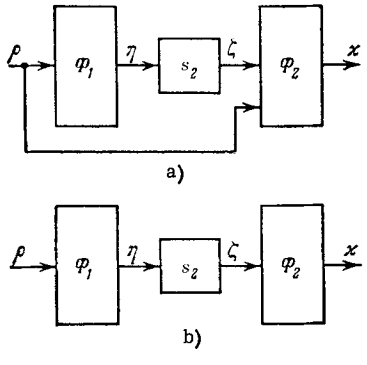


Fig. 4.3.

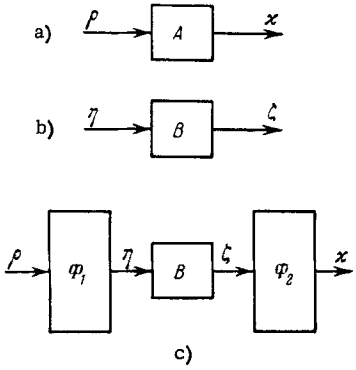


Fig. 4.4.

of the two converters performs functions of a single variable:

$$\eta = \Phi_1(\rho), \quad x = \Phi_2(\zeta).$$

In this special case, the problem formulated above has, in addition to a trivial solution, the following additional solution: the machine s_2 substitutes for the machine s_1 *if the state diagram of* s_1 *is superposable on the state diagram of* s_2 *(that is, is part of it) while preserving the uniqueness of functions* $\Phi_1(\rho)$ *and* $\Phi_2(\zeta)$. We shall illustrate this solution by an example.

Let us introduce the concept of *substitution for finite automata*, which is analogous to that of substitution for the s-machine: all the definitions are retained, except that instead of sequential machines s_1 and s_2, we are given two finite automata A and B (Fig. 4.4,a,b), and the substituting system is that of Fig. 4.4,c which is similar to that of Fig. 4.3,b.

(The above definitions obviously apply also to the special case of autonomous automata. However, the definition is simpler in this case since there are no input sequences and thus there is no need for an input symbol converter.)

As an example of the substitution of finite automata, let automata A and B have the state diagrams of Figs. 4.5 and 4.6, respectively. Then can automaton B, associated in the system of Fig. 4.4, substitute for automaton A? Converter Φ_2 relabels the states of automaton B (that is, it relabels the circles in its state diagram), while converter Φ_1 changes the labels above the arrows in the state diagram. (The usual condition of uniqueness must, of course, also be satisfied in the case of converters Φ_1 and Φ_2, that is, different values of the arguments should not result in the same value of the function.) If

after two such relabeling operations any part of the state diagram of automaton B still coincides with the state diagram of automaton A, then B substitutes for A.

In our example, the diagram of Fig. 4.5 is superposable on that part of Fig. 4.6 which consists of circles ζ_2, ζ_4 and ζ_5 with the associated arrows. Therefore converter $\varkappa = \Phi_2(\zeta)$ is a device for relabeling the states which operate in accordance with Table 4.1.

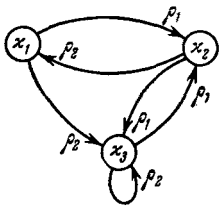


Fig. 4.5.

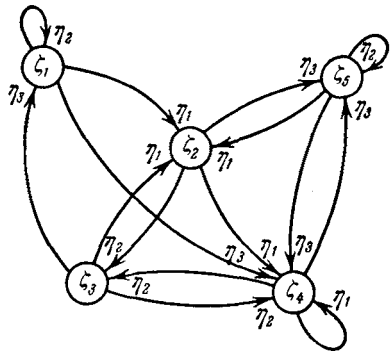


Fig. 4.6.

Table 4.1

ζ	ζ_1	ζ_2	ζ_3	ζ_4	ζ_5
$\varkappa = \Phi_2(\zeta)$		x_1		x_3	x_2

The operation of converter $\Phi_2(\zeta)$ is unspecified in states ζ_1 and ζ_3 of B, since these states do not occur during operation of the system. If desired, the operation of the converter in these states may be specified in an arbitrary manner, for instance, as shown in Table 4.2.

Table 4.2

ζ	ζ_1	ζ_2	ζ_3	ζ_4	ζ_5
$\varkappa = \Phi_2(\zeta)$	x_4	x_1	x_5	x_3	x_2

Converter $\Phi_2(\zeta)$ is now completely defined. It is seen from Table 4.2 that it satisfies the condition of uniqueness; that is, a given ζ uniquely determines $\varkappa$.

Let us now discuss converter $\Phi_1(\rho)$. The circles $\varkappa_1$ and $\varkappa_2$ in the diagram of A (Fig. 4.5) are connected by an arrow bearing the label ρ_1. Table 4.2 specifies that circle $\varkappa_1$ is matched by circle ζ_2 in the state diagram of automaton B (Fig. 4.6), and that circle $\varkappa_2$ of Fig. 4.5 is matched by circle ζ_5 in Fig. 4.6. The diagram of B shows that circles ζ_2 and ζ_5 are connected by an arrow labeled η_3 (that is, automaton B transforms from the state 1 into state 5 upon an input of η_3). Since automaton A transforms from state $\varkappa_1$ to $\varkappa_2$ upon an input of ρ_1, converter Φ_1 must place symbol η_3 into correspondence with symbol ρ_1.

Table 4.3

ρ	ρ_1	ρ_2
$\eta = \Phi_1(\rho)$	η_3	η_1

A similar reasoning may be applied to other portions of automatons A and B, and will finally result in Table 4.3 for converter Φ_1. This means that relationships

$$\Phi_1(\rho_1) = \eta_3, \quad \Phi_1(\rho_2) = \eta_1,$$

hold in every instance; that is, the condition of uniqueness is satisfied for converter Φ_1.

Now, if the labels above the branches of the state diagram of Fig. 4.6 were to be changed in accordance with Table 4.3 and the states were to be renumbered in accordance with Table 4.2, then the diagram of Fig. 4.6 would correspond exactly to a part of the diagram of Fig. 4.5.

To sum up, automaton A may be substituted by automaton B.

Still another variant of the same substitution is given by Tables 4.4 and 4.5 (instead of Tables 4.2 and 4.3).

Table 4.4

ζ	ζ_1	ζ_2	ζ_3	ζ_4	ζ_5
$\varkappa = \Phi_2(\zeta)$	$\varkappa_4$	$\varkappa_1$	$\varkappa_2$	$\varkappa_3$	$\varkappa_5$

Table 4.5

ρ	ρ_1	ρ_2
$\eta = \Phi_1(\rho)$	η_2	η_1

If the diagram of B were that of Fig. 4.7, the diagram of A remained the same (Fig. 4.5), then B could not substitute for A. Indeed, in this case there are only two possible tables for converter Φ_2: 4.6 or 4.7. However, the converter Φ_1 is not unique in these two cases since it is required that $\Phi_1(\rho_1) = \eta_3$ and at the same time that $\Phi_1(\rho_1) = \eta_2$. Substitution is therefore impossible. But this conclusion only holds for the substitution system of Fig. 4.4,c, because

Table 4.6

ζ	ζ_1	ζ_2	ζ_3	ζ_4	ζ_5
$\varkappa = \Phi_2(\zeta)$	$\varkappa_4$	$\varkappa_1$	$\varkappa_5$	$\varkappa_3$	$\varkappa_2$

Table 4.7

ζ	ζ_1	ζ_2	ζ_3	ζ_4	ζ_5
$\varkappa = \Phi_2(\zeta)$	$\varkappa_4$	$\varkappa_1$	$\varkappa_2$	$\varkappa_3$	$\varkappa_5$

such substitution becomes possible with the generalized system of Fig. 4.2. In that case, the converter Φ_2 is described by Table 4.6. The output $\varkappa$ depends on only one input ζ, that is, $\varkappa = \Phi_2(\xi)$, but the output of converter Φ_1 is a function of two variables, that is, $\eta = \Phi_1(\xi, \rho)$, and converter Φ_1 is described by Table 4.8.

Table 4.8

ρ \ ζ	ζ_2	ζ_4	ζ_5
ρ_1	η_3	η_2	η_3
ρ_2	η_1	η_1	η_1

Previously, we referred to "superposition of state diagrams." We meant by this not just the coincidence of the circles and arrows, and the labels above them, but also the coincidence of the positions of the labels above the arrows. In other words, we were concerned with substitution of automata and s-machines for automata and s-machines *of the same type.*

We showed in Chapter 3 that an s-machine of the P - Pr type could, as we expressed it, always be *transformed* into an s-machine of the P - P type. In using the term "transformation" we relied on the reader's intuitive grasp of this concept. Now, in the light of the definitions introduced in the present chapter, it is clear that in Chapter 3 we were in fact dealing with the substitution of s-machines of the P - P type for a machine of the P - Pr type.

4.2. THE ABSTRACT STRUCTURE OF THE AUTOMATON

In Chapter 3 the finite automaton was formally defined as an operator "processing" a sequence of symbols ρ into a sequence of symbols $\varkappa$ in such a way that the sequences do not contain contradictory triads. Put in these terms, the abstraction "finite automaton" is represented by the recurrent relationship

$$\left.\begin{aligned} \varkappa^{p+1} &= F(\varkappa^p, \rho^p) \quad \text{for type P - P automata,} \\ \varkappa^{p+1} &= F(\varkappa^p, \rho^{p+1}) \quad \text{for type P - N automata,} \end{aligned}\right\} \tag{4.1}$$

where F is any unique function defined on sets $\{\varkappa\}$ and $\{\rho\}$.

Now let us consider in detail how the description of a finite dynamic system such as an automaton of, for example, the P - P type, can be reduced to a relationship of the form (4.1).

Assume a finite dynamic system that has n generalized coordinates $x_1, x_2, \ldots, x_n$ and is subjected to s external (input) effects $u_1, u_2, \ldots, u_s$. We shall call each such input effect an ***input fiber***. At the sampling instants* $0, 1, \ldots, p, \ldots$, each of the generalized coordinates may assume only one of a finite number of values.

Assume coordinate x_i can have only one of k_i values ($i = 1, 2, \ldots, n$). Similarly, let each input effect u_j assume only one of r_j values ($j = 1, 2, \ldots, s$) at these instants. The "motion" in the system is subject to the condition that the value of each coordinate at the instant $p + 1$ must be uniquely determined by the values of all the coordinates x_i ($i = 1, 2, \ldots, n$) and of all the inputs u_j ($j = 1, 2, \ldots, s$) at the instant p. If that is the case, then the motion is described by the system of recurrence relations

$$x_i^{p+1} = f_i\left[x_1^p, x_2^p, \ldots, x_n^p; u_1^p, u_2^p, \ldots, u_s^p\right], \quad i = 1, 2, \ldots, n. \tag{4.2}$$

By introducing an n-dimensional vector $\boldsymbol{x}$ with coordinates $x_1, x_2, \ldots, x_n$, an s-dimensional vector $\boldsymbol{u}$ with coordinates $u_1, u_2, \ldots, u_s$, and a vector-function $\boldsymbol{f}$ with coordinates $f_1, f_2, \ldots, f_n$, relations (4.2) may be represented in vector form:

$$\boldsymbol{x}^{p+1} = \boldsymbol{f}\left[\boldsymbol{x}^p, \boldsymbol{u}^p\right]. \tag{4.3}$$

In the sampling instants, vector $\boldsymbol{x}$ can assume one of the $k = \prod_{i=1}^{n} k_i$ values, and the vector $\boldsymbol{u}$, one of the $r = \prod_{j=1}^{s} r_j$ values. Therefore, by selecting alphabets $\{\varkappa\}$ and $\{\rho\}$ consisting of k and r symbols, respectively, and assigning various symbols $\varkappa$ to the various vectors $\boldsymbol{x}$ and symbols ρ to the vectors $\boldsymbol{u}$, we obtain, instead of relation (4.3), a relation of the form of (4.1), in which there is a specific function F on the right-hand side. This function F is derived from the vector-function $\boldsymbol{f}$ (4.3) and is based on the coding selected for vectors $\boldsymbol{x}$ and $\boldsymbol{u}$.

It is now clear that the recurrence relations (4.2) represent a finite automaton

$$\varkappa^{p+1} = F(\varkappa^p, \rho^p). \tag{4.4}$$

*I.e., discrete moments.

Relations (4.2) illustrate more clearly than (4.4) such important features of a dynamic system as the number of its degrees of freedom, n, as well as the values of each of its generalized coordinates as a function of the state of each input fiber.

Let us agree to call a system of relations such as (4.2) the *abstract structure of the finite automaton* (4.4).

Thus a given abstract structure uniquely defines the corresponding finite automaton; that is, relation (4.4) may be uniquely reproduced from (4.2). In this sense (4.2) defines the automaton just as completely as (4.4) does. Accordingly, the concept of substitution of automata applies fully to machines defined by relations of type (4.2).

We shall now show that given a finite automaton A described by relation of type (4.4), we can specify a great number of abstract structures which can substitute for this automaton.

Let automaton A be associated with alphabets $\{\varkappa\}$ and $\{\rho\}$, and let k and r be given. We shall now select numbers n, s, k_i $(i = 1, 2, \ldots, n)$ and r_j $(j = 1, 2, \ldots, s)$. The selection of these numbers is restricted by only one condition: satisfaction of the inequalities

$$k \leqslant \prod_{i=1}^{n} k_i, \quad r \leqslant \prod_{j=1}^{s} r_j. \tag{4.5}$$

We now introduce n coordinates $x_1, x_2, \ldots, x_n$ (which assume $k_1, k_2, \ldots, k_n$ values, respectively) and s input fibers $u_1, u_2, \ldots, u_s$ (which assume $r_1, r_2, \ldots, r_s$ values, respectively). We now complete a table (Table 4.9) in the following manner.

Table 4.9

x^p				u^p				$\varkappa^p$	ρ^p	$\varkappa^{p+1}$	x^{p+1}			
x_1	x_2	...	x_n	u_1	u_2	...	u_s				x_1	x_2	...	x_n

We enter all the possible combinations of values $x_1, x_2, \ldots, x_n$ and $u_1, u_2, \ldots, u_s$ into the left-hand columns x^p, and u^p of the table. The number of such combinations is $\prod_{i=1}^{n} k_i \prod_{j=1}^{s} r_j$, and therefore Table 4.9 shall contain this number of rows.

In order to fill in column $\varkappa^p$ let us concentrate exclusively on columns x^p. We enter in these columns $\prod_{i=1}^{n} k_i$ different combinations of $x_1, x_2, \ldots, x_n$. Let us select at random k of these combinations, which we shall call the basic combinations, assign to them symbols $\varkappa_1, \varkappa_2, \ldots, \varkappa_k$, and enter these symbols into the corresponding rows of the $\varkappa^p$ column of Table 4.9. By virtue of inequalities (4.5) it is possible that $\prod_{i=1}^{n} k_i > k$; therefore some of the rows in column $\varkappa^p$ may remain blank. If that is the case, we assign to those combinations of $x_1, x_2, \ldots, x_n$ which were not included in the k selected combinations the used symbols $\varkappa$ (the order of assignment is immaterial), and enter these symbols into the blank rows of column $\varkappa^p$. Now column $\varkappa^p$ is completely filled in. We fill column ρ^p in a similar manner, using combinations of $u_1, u_2, \ldots, u_s$ entered in columns u^p.

At the end of this procedure, columns $\varkappa^p$, ρ^p contain all the possible combinations of the symbols $\varkappa$, ρ, but since the total number of such combinations is only

$$kr \leqslant \prod_{i=1}^{n} k_i \prod_{j=1}^{s} r_j,$$

some combinations of $\varkappa$, ρ may recur.

We now return to our automaton A. Using one of its definitions, for example, its basic table, we fill in column $\varkappa^{p+1}$ of Table 4.9. But we have already associated one of the basic combinations of $x_1, x_2, \ldots, x_n$ with each symbol $\varkappa$. We therefore enter in columns x^{p+1} the combinations corresponding to $\varkappa^{p+1}$, thus completing the table. This table defines the values of all the x_m^{p+1} starting from the given x_i^p and u_j^p, that is, it defines n functions f_i in recurrence relations (4.2).

If inequalities (4.5) were to be replaced by equations

$$k = \prod_{i=1}^{n} k_i, \quad r = \prod_{j=1}^{s} r_j, \tag{4.6}$$

then each pairwise combination of symbols $\varkappa$, ρ would be encountered only once in columns $\varkappa^p$, ρ^p of Table 4.9. If, however,

$$k < \prod_{i=1}^{n} k_i, \quad r = \prod_{j=1}^{s} r_j, \tag{4.7}$$

then Table 4.9 would contain the same rows as in the case in which

(4.6) holds, and, in addition, some supplementary rows, corresponding to the nonbasic combinations.

If we now attempted to derive an automaton from the abstract structure of Table 4.9, we would obtain an automaton differing from the starting one. The state diagram of such a new automaton would contain all the circles and branches of the diagram of the starting machine (these would be defined by the rows containing the basic combinations of x and u), but it would also contain supplementary circles and branches (corresponding to the nonbasic combinations of x and u). Since the conditions requiring unique operation of symbol converters are satisfied (by the very construction), then, assuming case (4.7) holds, the abstract structure defines an automaton that substitutes for the given (starting) automaton.

When we set up the abstract structure, that is, constructed the system of relations (4.2) from relation (4.4) with the aid of Table 4.9, we had no restriction on the selection of numbers n, s, k_i and r_j, provided conditions (4.5) were satisfied. It is obvious now that not only the form of the functions f_i on the right-hand side of (4.2) but also the number of relations involved in that system depend on how these numbers have been selected. And it is because we have this freedom that we can construct a large number of abstract structures, all which substitute for a given finite automaton A.

An important special case is one in which all the k_i and r_j equal two, that is, all the x and u are logical variables. The abstract structure in this case is

Table 4.10

χ \ p	p_1	p_2	p_3
χ_1	χ_3	χ_4	χ_6
χ_2	χ_2	χ_5	χ_4
χ_3	χ_6	χ_2	χ_3
χ_4	χ_5	χ_1	χ_5
χ_5	χ_3	χ_1	χ_2
χ_6	χ_1	χ_3	χ_2

$$x_i^{p+1} = L_i\left[x_1^p, x_2^p, \ldots, x_n^p; u_1^p, u_2^p, \ldots, u_s^p\right], \quad i = 1, 2, \ldots, n, \tag{4.8}$$

where all the L_i are logical functions. We shall call such an abstract structure ***logical*** or ***binary***. In this case,

$$\prod_{i=1}^{n} k_i = 2^n \text{ and } \prod_{j=1}^{s} r_j = 2^s. \tag{4.9}$$

If k and r are not integral powers of 2 (that is, they are not among

numbers 2, 4, 8, 16, 32, 64, 128, 256, ...), then Eq. (4.6) cannot be satisfied. To satisfy inequalities (4.5), n and s must be selected in accordance with the conditions

$$k < 2^n \text{ and } r < 2^s.$$

Thus, for example, if the automation is defined (given) by its basic table (Table 4.10), then $k = 6$, $r = 3$, and we can select, for example, $n = 3$, $s = 2$. The completion of a table such as Table 4.9 for this case is illustrated in Table 4.11.

Table 4.11

x^{p-1}			u^{p-1}		χ^{p-1}	ρ^{p-1}	γ^p	x^p		
x_1	x_2	x_3	u_1	u_2				x_1	x_2	x_3
0	0	0	0	0	χ_1	ρ_1	χ_3	0	1	0
0	0	1	0	0	χ_2	ρ_1	χ_2	0	0	1
0	1	0	0	0	χ_3	ρ_1	χ_6	1	0	1
0	1	1	0	0	χ_4	ρ_1	χ_5	1	0	0
1	0	0	0	0	χ_5	ρ_1	χ_3	0	1	0
1	0	1	0	0	χ_6	ρ_1	χ_1	0	0	0
1	1	0	0	0	χ_1	ρ_2	χ_4	0	1	1
1	1	1	0	0	χ_2	ρ_2	χ_5	1	0	0
0	0	0	0	1	χ_3	ρ_2	χ_2	0	0	1
0	0	1	0	1	χ_4	ρ_2	χ_1	0	0	0
0	1	0	0	1	χ_5	ρ_2	χ_1	0	0	0
0	1	1	0	1	χ_6	ρ_2	χ_3	0	1	0
1	0	0	0	1	χ_1	ρ_3	χ_6	1	0	1
1	0	1	0	1	χ_2	ρ_3	χ_4	0	1	1
1	1	0	0	1	χ_3	ρ_3	χ_3	0	1	0
1	1	1	0	1	χ_4	ρ_3	χ_5	1	0	0
0	0	0	1	0	χ_5	ρ_3	χ_2	0	0	1
0	0	1	1	0	χ_6	ρ_3	χ_2	0	0	1
0	1	0	1	0	χ_1	ρ_1	χ_3	0	1	0
0	1	1	1	0	χ_1	ρ_1	χ_3	0	1	0
1	0	0	1	0	χ_1	ρ_1	χ_3	0	1	0
1	0	1	1	0	χ_1	ρ_1	χ_3	0	1	0
1	1	0	1	0	χ_1	ρ_1	χ_3	0	1	0
1	1	1	1	0	χ_1	ρ_1	χ_3	0	1	0
0	0	0	1	1	χ_1	ρ_1	χ_3	0	1	0
0	0	1	1	1	χ_1	ρ_1	χ_3	0	1	0
0	1	0	1	1	χ_1	ρ_1	χ_3	0	1	0
0	1	1	1	1	χ_1	ρ_1	χ_3	0	1	0
1	0	0	1	1	χ_1	ρ_1	χ_3	0	1	0
1	0	1	1	1	χ_1	ρ_1	χ_3	0	1	0
1	1	0	1	1	χ_1	ρ_1	χ_3	0	1	0
1	1	1	1	1	χ_1	ρ_1	χ_3	0	1	0

We can also develop other binary structures that substitute for the same automaton; thus, for instance, we can take $n = 4$, $s = 3$, or $n = 3$, $s = 3$, and so on.

Table 4.12

x^p		u^p	$\varkappa^p$	ρ^p	$\varkappa^{p+1}$	x^{p+1}	
x_1	x_2					x_1	x_2
0	0	0	$\varkappa_1$	ρ_1	$\varkappa_3$	0	2
0	1	0	$\varkappa_2$	ρ_1	$\varkappa_2$	0	1
0	2	0	$\varkappa_3$	ρ_1	$\varkappa_6$	1	2
1	0	0	$\varkappa_4$	ρ_1	$\varkappa_5$	1	1
1	1	0	$\varkappa_5$	ρ_1	$\varkappa_3$	0	2
1	2	0	$\varkappa_6$	ρ_1	$\varkappa_1$	0	0
0	0	1	$\varkappa_1$	ρ_2	$\varkappa_4$	1	0
0	1	1	$\varkappa_2$	ρ_2	$\varkappa_5$	1	1
0	2	1	$\varkappa_3$	ρ_2	$\varkappa_2$	0	1
1	0	1	$\varkappa_4$	ρ_2	$\varkappa_1$	0	0
1	1	1	$\varkappa_5$	ρ_2	$\varkappa_1$	0	0
1	2	1	$\varkappa_6$	ρ_2	$\varkappa_3$	0	2
0	0	2	$\varkappa_1$	ρ_3	$\varkappa_6$	1	2
0	1	2	$\varkappa_2$	ρ_3	$\varkappa_4$	1	0
0	2	2	$\varkappa_3$	ρ_3	$\varkappa_3$	0	2
1	0	2	$\varkappa_4$	ρ_3	$\varkappa_5$	1	1
1	1	2	$\varkappa_5$	ρ_3	$\varkappa_2$	0	1
1	2	2	$\varkappa_6$	ρ_3	$\varkappa_2$	0	1

We shall now show that the same automaton could be replaced by an abstract structure that is not binary. Suppose, for example, that $n = 2$, $k_1 = 2$, $k_2 = 3$, but that $s = 1$, $r_1 = 3$ as before. Then $k = k_1 k_2 = 6$, and $r = r_1 = 3$; that is, Eqs. (4.6) are satisfied.

Completing Table 4.9 for this case, we get an abstract structure of Table 4.12, but this structure is no longer binary.

4.3. NETS

Suppose we have the simplest finite automaton, for which Eq. (4.1) becomes

$$\varkappa^{p+1} = \rho^p. \tag{4.10}$$

In analyzing this automaton we assume that even if the alphabet $\{\varkappa\}$ contains some symbols that are not contained in $\{\rho\}$, these shall appear only at the initial instant. The symbol $\varkappa$ produced by such an automaton at the instant p is identical to the input symbol in the preceding instant $p - 1$. We shall call this simple automaton a *one-instant delay* (or simply *delay*).

Returning to our P - P automaton and introducing a new variable μ (alphabet $\{\mu\}$ coincides with alphabet $\{\varkappa\}$), and replace (4.1) by relations

$$\left.\begin{aligned} \mu^p &= F[\varkappa^p, \rho^p], \\ \varkappa^p &= \mu^{p-1}. \end{aligned}\right\} \tag{4.11}$$

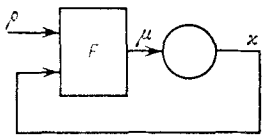


Fig. 4.8.

Such a relationship was already discussed in Section 3.3. We shall now treat the first of relations (4.11) as representing an instantaneous converter of the symbols $\varkappa$ and ρ into symbols μ, while the second relation defines a delay. Accordingly, relations (4.11) may be represented by the system of Fig. 4.8, where the delay is shown as a circle.

Let us now consider the abstract structure of some automaton, for instance, of a P - P automaton, that is, a system of relations of the form (4.2):

$$\begin{gathered} x_i^{p+1} = f_i[x_1^p, x_2^p, \ldots, x_n^p; u_1^p, u_2^p, \ldots, u_s^p], \\ i = 1, 2, \ldots, n. \end{gathered} \tag{4.12}$$

Since each relation of this system, for example,

$$x_1^{p+1} = f_1[x_1^p, x_2^p, \ldots, x_n^p; u_1^p, u_2^p, \ldots, u_s^p],$$

may in itself be treated as combining the delay

$$x_1^{p+1} = y_1^p$$

and the function converter

$$y_1 = f_1[x_1, x_2, \ldots, x_n; u_1, u_2, \ldots, u_s] \tag{4.13}$$

the entire abstract structure (4.12) may be represented by n delays and n function converters. The input of each delay is connected to the output of the corresponding converter. The outputs of all the delays are connected to the inputs of each of the converters. In addition, we feed external effects u to the inputs of all the converters (Fig. 4.9).*

If we have several abstract structures, we can derive new abstract structures by "interconnecting" the original ones.

*In Fig. 4.9 and henceforth, circles represent delays and rectangles denote converters.

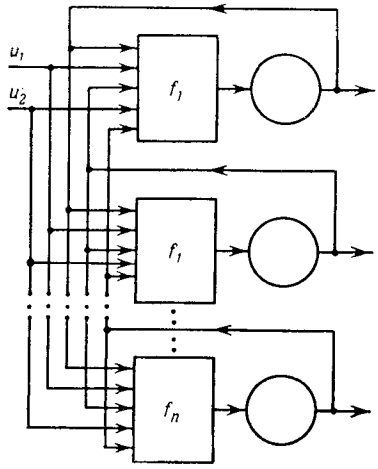


Fig. 4.9.

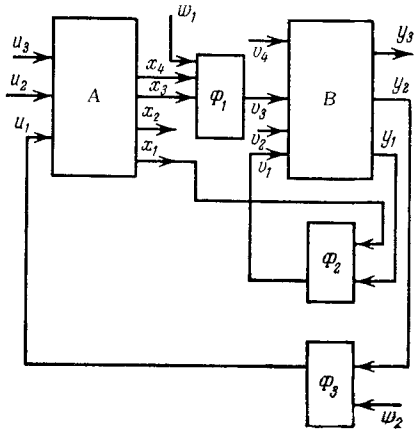


Fig. 4.10.

By "interconnection of automata" we mean the identifying of the output symbols of one automaton with the input symbols of another automaton. In this sense the output of one automaton can act upon the input of another one only if all the symbols of the output alphabet of the first automaton are contained in the input of the second. If this is not the case then "interconnection" of automata can be achieved only by means of auxiliary converters.

For example, suppose we have two abstract structures: structure A

$$x_i^{p+1} = f_i(x_1^p, x_2^p, x_3^p, x_4^p; u_1^p, u_2^p, u_3^p), \quad i = 1, 2, 3, 4 \tag{4.14}$$

and structure B

$$y_j^{p+1} = \varphi_j(y_1^p, y_2^p, y_3^p, v_1^p, v_2^p, v_3^p, v_4^p), \quad j = 1, 2, 3. \tag{4.15}$$

Let us supplement (4.14) and (4.15) with the equations of three converters:

$$\left.\begin{aligned} v_3 &= \Phi_1(x_3, x_4, w_1), \\ v_1 &= \Phi_2(y_1, x_1), \\ u_1 &= \Phi_3(y_2, w_2). \end{aligned}\right\} \tag{4.16}$$

Here w_1 and w_2 are auxiliary input fibers of the converters. Figure 4.10 shows schematically the coupling of structures A and B by means of these three converters.

Together, Eqs. (4.14), (4.15), and (4.16) again give an abstract structure. Indeed, substituting (4.16) into (4.14) and (4.15), we obtain a system of seven recurrence relations with coordinates (x_1, x_2, x_3, x_4, y_1, y_2 and y_3) and input fibers (u_2, u_3, v_2, w_1 and w_2), of the same type which we already know as an "abstract structure."

We shall designate by the term *net* a system of finite number of recurrence relations similar to Eq. (4.2) and supplemented by converter equations which express some of the inputs by means of coordinates.

The net itself is an abstract structure. Its coordinates are all the generalized coordinates of all its component abstract structures. The input fibers of the net can be both the input fibers of the component abstract structures that are not acted upon by converters, as well as the input fibers of the converters that are not acted upon by the coordinates of the abstract structures constituting the net.

To obtain a system of relations such as (4.2) for the net, one uses the converter equations to eliminate the input variables acted upon by the converters.

Since the net itself is an abstract structure, it substitutes for some finite automaton. Thus when one uses converters to combine abstract structures into nets, one generates new finite automata from other finite automata.

We shall say that a net is a *delay net* if it consists only of delays connected by means of function converters. It follows from previous discussion that any abstract structure and any net can be represented by a delay net. This was shown in Fig. 4.9. Such a representation is not unique in the sense that every delay net may substitute for some automaton, and, as was pointed out in the preceding section, one can generate many abstract structures which can substitute for each automaton. This means that one can construct many

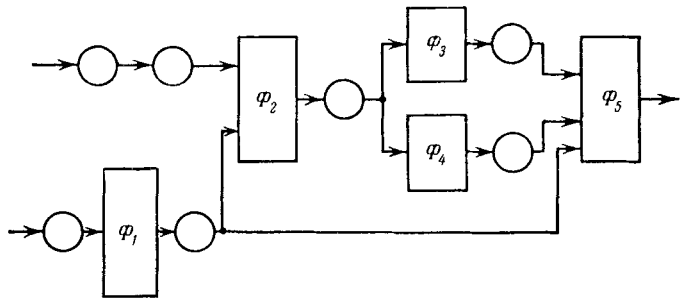


Fig. 4.11.

delay nets to represent the same automaton, such nets differing in the number of constituent delays and in the alphabets on which the delays are defined.

Among these delay nets we can distinguish the subclass of *loop-free nets*. A delay net is said to be a loopfree net if, starting from any delay, one moves along the net in the direction of its operation (along the arrows of the schematic such as Fig. 4.5) and never returns to the starting delay. Figure 4.11 shows a loopfree net, while Fig. 4.12 presents a net containing loops.

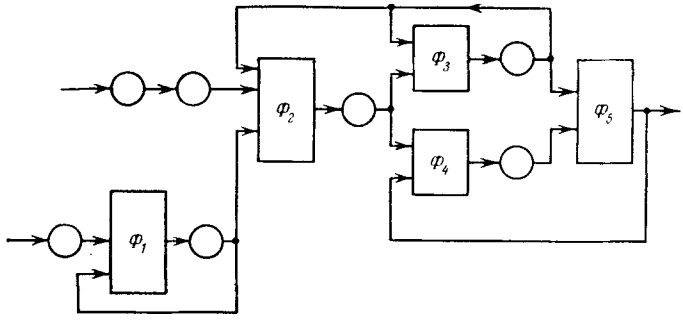


Fig. 4.12.

One important loop-free net is that consisting of q series-connected delays (Fig. 4.13). If the inputs and outputs of the delays are defined on different alphabets, then one must interpose converters between the delays (Fig. 4.14).

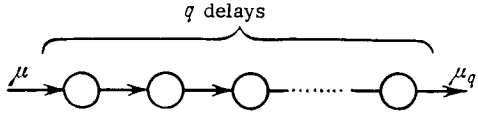


Fig. 4.13.

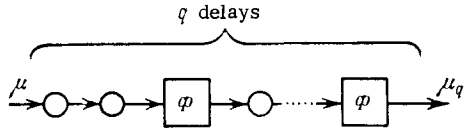


Fig. 4.14.

Let us call such a net a *delay line*. As any other net, a delay line is a finite automaton, but with an important difference. If the delay line has an input μ and an output μ_q, then the symbol μ_q will not be a function of the symbol μ generated during the preceding sampling instant, but will be defined by the input symbol μ appearing q instants before:

$$\mu_q^{p+q} = f(\mu^p).$$

If the input and output symbols of all the delays are defined on the same alphabet, then output μ_q will coincide with input μ, supplied q instants before:

$$\mu_q^{p+q} = \mu^p.$$

Let us now return to the diagram of Fig. 4.8, but let us substitute its delay by a delay line. We then obtain Fig. 4.15. Then, instead of relations (4.11), we have

$$\left.\begin{aligned} \mu^p &= F(x^p, \rho^p), \\ x^{p+q} &= \mu^p. \end{aligned}\right\} \tag{4.17}$$

Eliminating μ, we have

$$x^{p+q} = F(x^p, \rho^p), \tag{4.18}$$

whereas by eliminating x from (4.17) [rather than μ], we obtain

$$\mu^{p+q} = F(\mu^p, \rho^{p+q}). \tag{4.19}$$

A delay line can also be connected to the input of a converter. Then, instead of relations (4.17), we get

$$\left.\begin{aligned} \mu^p &= F(x^p, \psi^p), \\ \psi^{p+q} &= \rho^p, \\ x^{p+q} &= \mu^p, \end{aligned}\right\} \tag{4.20}$$

or, by eliminating ψ and x,

$$\mu^{p+q} = F(\mu^p, \rho^p). \tag{4.21}$$

Note that if we had designated by symbols z the state of the entire system, taking into account the outputs of all the delays, then

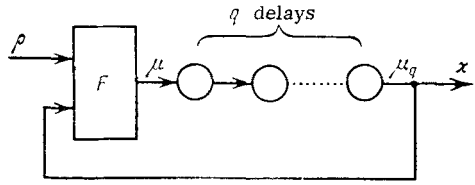


Fig. 4.15.

at the instant $p + 1$ the $\varkappa$ would depend only on the $\varkappa$ at the instant p. However, if we assign the symbol $\varkappa$ only to the output of the last delay, we can use this automaton to embody relation (4.19). This is precisely what we meant when, on introducing the concept of a finite automaton in Chapter 3, we said that an expression such as (3.5′):

$$\varkappa^{p+1} = F(\varkappa^p, \rho^p)$$

is so general that it includes a finite automaton in which the output $\varkappa^{p+q}$ depends on the inputs $\varkappa^p$ and ρ^p introduces a finite number (q) of instants previously.

Now let us return to the relationship between automata and sequential machines of the P - P and P - Pr types. Consider the schematic of Fig. 4.8, described by Eq. (4.11). As stated in Chapter 3, Eq. (4.11) describes automata of either the P - P or P - Pr type, depending on whether we eliminate variable μ or variable $\varkappa$. The diagram of Fig. 4.8 can represent either case, depending on whether the output variable is $\varkappa$ (P - P automaton) or μ (P - Pr automaton). Figures 4.16,a and b shows these two machines.

As pointed out in Chapter 3, a P - P s-machine differs from P - Pr machine only in that it contains a P - P rather than a P - Pr automaton. Thus such machines may be represented as shown in Fig. 4.17,a,b.

Let us recall that the P - P machine can always substitute for a P - Pr machine simply by changing converter Φ. However, the converse statement, that is, that merely by exchanging the output converter a P - Pr machine can be made to substitute for a P - P machine, is not true. Now that we can define automata by means of tables and have introduced diagrams such as Figs. 4.16 and 4.17, we can illustrate this statement by an example.

Assume an s-machine of the P - P type (Fig. 4.17,a) is given by the tables of converter F (Table 4.13) and converter Φ (Table 4.14).

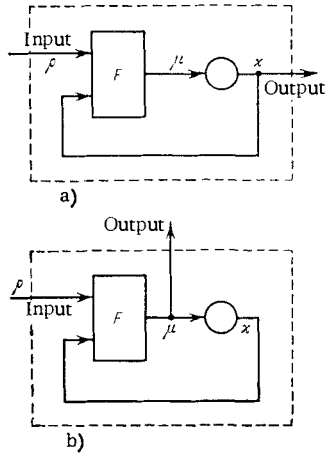


Fig. 4.16.

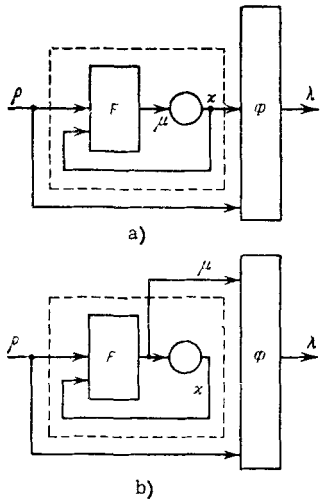


Fig. 4.17.

Table 4.13

Converter F

x \ ρ	ρ_1	ρ_2	ρ_3
x_1	x_3	x_1	x_2
x_2	x_3	x_3	x_4
x_3	x_2	x_1	x_2
x_4	x_2	x_4	x_3

Table 4.14

Converter Φ

x \ ρ	ρ_1	ρ_2	ρ_3
x_1	λ_2	λ_3	λ_2
x_2	λ_3	λ_1	λ_3
x_3	λ_2	λ_1	λ_2
x_4	λ_1	λ_2	λ_3

Retaining converter F as is, find a new table for a converter $\lambda = \Phi^*(\mu, \rho)$ such that the s-machine Fig. 4.17, b incorporating it will substitute for the initial s-machine.

Consider, for example, the intersection of the first row and the first column in the table for converter Φ (Table 4.14), where one finds symbol λ_2. The corresponding square of Table 4.13 for converter F contains the symbol x_3. We shall thus write the symbol λ_2 at the intersection of the first column and the third row in the table of the new converter Φ^* (Table 4.15): we can complete the remainder of this table in the same manner.

Now, however, let us try to repeat the procedure, this time starting with the square located in the first column and the second row of Table 4.14 of the old converter Φ. This square contains symbol λ_3, and the corresponding square of Table 4.13 contains the symbol $\varkappa_3$ as before.

Thus our new table would have to contain λ_3 in a square already occupied by symbol λ_2. But this means that we would obtain a nonunique converter. Our example confirms the statement that, in general, a P - Pr machine cannot substitute for a P - P machine if the only change introduced into the P - Pr machine is that of the output converter.

Table 4.15

Converter Φ^*

$\varkappa$ \ p	p_1	p_2	p_3
$\varkappa_1$			
$\varkappa_2$			
$\varkappa_3$	λ_2		
$\varkappa$			

Now let us return to the relationship between a finite automaton and a sequential machine. We have accumulated sufficient material to define this relationship more precisely, by means of the following theorem.*

Theorem. *For every sequential machine s there exists a system consisting of a finite automaton A and an output converter such that, given any initial state of S and any input sequence to it, there is an initial state of A such that, at all $p \geqslant 1$, the output sequence of A is a repetition of the output sequence of S with a delay of one sampling instant. Conversely, for any system consisting of a finite automaton A and an output converter, there exists a sequential machine s such that, given an initial state of A and any input sequence to it, there is an initial state of S such that, at all $p \geqslant 0$ the output sequence of S repeats the output sequence of A with a lead of one sampling instant.*

Proof of the first statement. Assume a sequential machine s described by equations

$$\varkappa^{p+1} = F(\varkappa^p, p^p), \tag{4.22}$$

$$\lambda^p = \Phi(\varkappa^p, p^p). \tag{4.23}$$

We shall now construct a net consisting of two finite automata and define it by equations

$$y^{p+1} = F_1(y^p, z^p), \tag{4.24}$$

$$z^{p+1} = p^p, \tag{4.25}$$

*Compare this with Theorem 1 of [16].

and we define the output converter of the net by

$$\chi^p = \Phi(y^p, z^p), \tag{4.26}$$

where the alphabets $\{y\}$ and $\{\chi\}$ coincide with the alphabets $\{\varkappa\}$ and $\{\lambda\}$, respectively; the alphabet $\{z\}$ differs from $\{\rho\}$ by one additional symbol z_0; the function Φ in (4.26) coincides with the corresponding function in (4.23) for all pairs of symbols from alphabets $\{\varkappa\}$ and $\{\rho\}$, but is indeterminate (or may be defined in any desired way) for $z = z_0$; the function F_1 in (4.24) coincides with F in (4.22) for all combinations of symbols that do not contain z_0, but for z_0-containing combinations, $F_1(y, z_0) = y$. Equations (4.24) and (4.25) specify the finite automaton A whose states are coded by the symbol pair (y, z), while Eq. (4.26) defines the output converter for this automaton. We shall now compare with each initial state $\varkappa^0$ of S that initial state of automaton A for which

$$y^0 = \varkappa^0, \quad z^0 = z_0. \tag{4.27}$$

When $p = 0$, it follows from (4.24) that $y^1 = y^0 = \varkappa^0$. When $p \geqslant 1$, the symbol z^0 cannot be generated, so that for the sampling instants the function F_1 in (4.24) can be replaced by F, and

$$y^{p+1} = F(y^p, z^p), \quad p \geqslant 1. \tag{4.28}$$

Introducing a new variable $Y^p \equiv y^{(p+1)}$ $(p \geqslant 0)$, and using (4.24), we can write

$$Y^{p+1} = F(Y^p, \rho^p), \quad p \geqslant 0, \tag{4.29}$$

whereby $Y^0 = y^1 = \varkappa^0$. Equations (4.22) and (4.29) and the initial conditions $\varkappa(0) = \varkappa^0$ coincide, so that for any $p \geqslant 1$

$$Y^p \equiv \varkappa^p.$$

But $y^p = Y^{p-1}$, so that

$$y^p = \varkappa^{p-1}, \quad p \geqslant 1. \tag{4.30}$$

Substituting (4.25) and (4.30) into (4.26) and comparing with (4.23), we get

$$\chi^p = \Phi(\varkappa^p, \rho^p) = \lambda^{p-1}, \quad p \geqslant 1,$$

which proves the first statement of the theorem.

We shall now prove the second statement. Let the finite automaton A be

$$z^{p+1} = F(z^p, \rho^p) \tag{4.31}$$

and its output converter

$$\chi^p = \Phi(z^p). \tag{4.32}$$

We shall now compare this automaton with the sequential machine s:

$$y^{p+1} = F(y^p, \rho^p), \tag{4.33}$$

$$\lambda^p = \Phi[F(y^p, \rho^p)], \tag{4.34}$$

where the alphabets $\{y\}$ and $\{z\}$ coincide. We shall match each initial state of A with an initial state of S. Since Eqs. (4.31) and (4.33) and the corresponding initial conditions coincide, we have

$$y^p = z^p \tag{4.35}$$

for all $p \geqslant 0$. Substituting (4.35) into (4.34) and using (4.31) and (4.32), we have, for all $p \geqslant 0$,

$$\lambda^p = \Phi[F(z^p, \rho^p)] = \Phi(z^{p+1}) = \chi^{p+1},$$

q. e. d. Thus the entire theorem is proved.

If one wants to determine what a sequential machine or a finite automaton "can do," then, by virtue of the above theorem, one need not examine these machines separately. However, the two abstract concepts are not equivalent, in the sense that the same "task" may be performed in a sequential machine with a smaller number of states than in the corresponding automaton. This is important in cases in which it is desired to minimize the number of such states (see Chapter 9).

4.4. ABSTRACT AGGREGATES OF AUTOMATA AND SEQUENTIAL MACHINES

We have proved that an automaton may be replaced by a variety of abstract structures, and that each abstract structure may itself be replaced by a great variety of nets. On the other hand, if each abstract structure is an automaton, then the very concept of a net

permits us to devise new automata from other automata. This reasoning leads to the following problem: is it possible to find such a set of automata and converters that, by employing any devised number of automata and converters of this set, one can construct nets which would substitute for a variety of automata and sequential machines?

The process of constructing nets by employing only automata and converters of a given set will be called the *aggregation of finite automata and sequential machines.*

Automata and converters contained in a set are said to be *elements of the set.* We shall say that a set is *complete* if its elements can be used to construct a net which can substitute for any *a priori* given finite automaton or sequential machine.

We have already shown that, given a set of any delay elements (that is, operating in any alphabets) and any converters, one can construct a net which will substitute for any given automaton.

Assume a delay element operating in alphabet $\{\mu\}$. If, in addition, we had some set of elementary converters from which we could construct any converter operating in alphabet $\{\mu\}$, then we would have a complete set.

Thus, for example, one important complete set is that consisting of a binary delay element (that is, a bistable delay element) plus elements performing the operations of disjunction, conjunction, and negation.

Indeed, any automaton may be replaced by an abstract logical structure, that is, an abstract structure in which all the $k_i = 2$ and all the $r_j = 2$ $(i = 1, 2, \ldots, n;\ j = 1, 2, \ldots, s)$. But such an abstract structure may be represented by a net consisting exclusively of binary delay elements and logical converters. But since any logical function may be represented by a conjunction of disjunctive groups, any logical converter can be constructed of converters performing disjunction, conjunction, and negation (see Chapter 1 and 2). Therefore, the above set is complete.

Obviously, we would also have a complete set if the latter consisted of the binary delay element plus a converter performing any logical function (such as a converter performing the Sheffer stroke).

In a similar manner, one can develop complete sets operating in alphabets containing more than two symbols (for instance, m symbols); the problem then reduces to that of expressing any logical function of m-valued logic by means of several primitive functions. Such primitive functions, plus a delay element operating in the alphabet of m symbols, constitute a complete set.

Chapter 5 will describe the practical embodiments of various complete sets as well as the construction of finite automata and

sequential machines based on such sets. However, let us first briefly consider an important abstract model, which was developed in connection with certain problems in physiology.

4.5. ABSTRACT NEURONS AND MODELS OF NEURAL NETS

The behavior of nerve cells (neurons) and of nervous systems is assumed to be representable by abstract (idealized) neurons and abstract models of neural nets. The McCulloch - Pitts neural net is one of the best known abstractions of this kind.*

The *McCulloch-Pitts neuron* is an imaginary logical element which may exist in only one of two possible states:

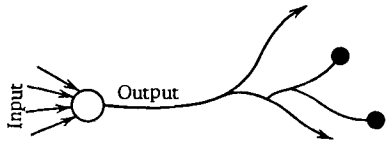


Fig. 4.18.

"stimulated" and "not stimulated." A neuron may have any finite number of inputs, but only one output which may, however, have any finite number of branches (Fig. 4.18). Each input may terminate in either of two endings: *"inhibitory"* (black dot in Fig. 4.18) or *"simple"* (arrow in Fig. 4.18). The branching output endings of a neuron may act upon the inputs of other neurons or on their own input. Some of the neuronal inputs may be externally stimulated. Again, an external stimulus may either stimulate or not stimulate the input. We can thus form *abstract neural nets* to simulate nerve tissues (see Fig. 4.19).

Let $f(t)$ be the number of simple input ("$\rightarrow$") endings which are stimulated at the instant t and act upon a given neuron and let $g(t)$ be the number of stimulated inhibitory endings ($\bullet$) which also act upon this neuron.

The functioning of a neuron (and, consequently, of the net) is determined by the following condition of stimulation: A neuron is stimulated at the instant $t+\tau$ if the following conditions are satisfied at the instant t:

$$g(t)=0, \quad f(t)\geqslant h, \tag{4.36}$$

*Developed in 1943 (see [62]), this abstraction has by now lost its value to physiology because of more recent studies in the properties of neurons.

where h is a given finite number, called the *threshold of stimulation.* A neuron is not stimulated unless these conditions are met. Thus the inhibitory ending has "veto power"; that is, even if inequality (4.36) is satisfied, the output is not stimulated if the input of the neuron in question is connected to even one inhibitory branch of a stimulated neuron.*

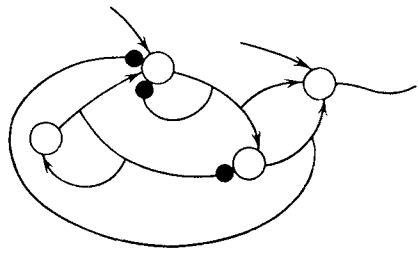


Fig. 4.19.

If w_i is the state of the simple input fibers ($i = 1, 2, \ldots, s$), v_j is the state of the inhibitory input fibers ($j = 1, 2, \ldots, q$), and x is the state of the neuron, then the behavior of a McCulloch-Pitts neuron, such that $s = 3$, $h = 2$, $q = 2$ is described by

$$x(t+\tau) = \{[w_1(t) \& w_2(t)] \vee [w_1(t) \& w_3(t)] \vee [w_2(t) \& w_3(t)]\} \& \overline{v_1(t)} \& \overline{v_2(t)}. \tag{4.37}$$

Let us mark off points $0, \tau, 2\tau, 3\tau, \ldots$ on the time axis and observe the neurons and neural nets only at such instants; that is, let us introduce sampling instants. Instead of $0, \tau, 2\tau, 3\tau, \ldots$ we introduce integers $0, 1, 2, \ldots, p \ldots$ respectively, to denote the occurrence of these sampling instants. Then expression (4.37) may be written as

$$x^{p+1} = [(w_1^p \& w_2^p) \vee (w_1^p \& w_3^p) \vee (w_2^p \& w_3^p)] \& \overline{v_1^p} \& \overline{v_2^p}. \tag{4.38}$$

Now consider any McCulloch-Pitts abstract neural net consisting of n neurons and having s "free" inputs, through which external effects can be introduced. We shall label the neurons contained in the net by $1, 2, \ldots, n$ and the free input fibers by $1, 2, \ldots, s$; we shall also denote by x_i the state of the ith neuron and by u_j the state of the jth input fiber of the net. Then we can write n equations of the form (4.38). Each input of a neuron of the net is acted upon either

*Frequently, other conditions for the functioning of neurons are specified (see, for example, [73]).

by the ending of one of the other neurons or by an external effect. Therefore each w or v can be identified with an x or a u; that is, we can introduce x^p or u^p into the right-hand sides of expressions (4.38), to replace w^p and v^p, respectively.

It follows from the foregoing that the McCulloch-Pitts abstract neural net can be described by the system of relations

$$x_i^{p+1} = L_i(x_1^p, x_2^p, \ldots, x_n^p; u_1^p, u_2^p, \ldots, u_s^p), \quad i = 1, 2, \ldots, n, \tag{4.39}$$

where L_i are logical functions such as (4.38).

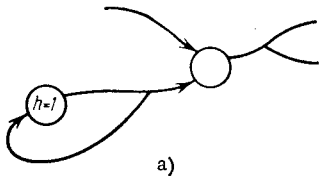
Thus, the McCulloch-Pitts abstract neural net is in effect a net according to our definition of this term, and is therefore a finite automaton. But since the right-hand side of (4.39) contains not just any logical functions but functions of a special type [the (4.38) type], there arises a question: Can we construct a neural net to correspond to any given finite automaton operating in the 0, 1, 2, ... (that is, 0, τ, 2τ, ...) timing sequence?

To answer this question, it will be pointed out, first of all, that a self-simulated neuron with $h = 1$ and no inhibiting inputs (Fig. 4.20) is a blocked, or permanently simulated, neuron. Therefore neurons may have permanently stimulated inputs (Fig. 4.21,a), denoted as shown in Fig. 4.21,b. We shall say in this case that the stimulated input fiber is *fixed.*

We shall now consider a neuron with $h = 1$, $s = 1$, and $q = 0$ (Fig. 4.22). Such a neuron is described by

$$x^{p+1} = w^p,$$

that is, one neuron will produce a delay



a)

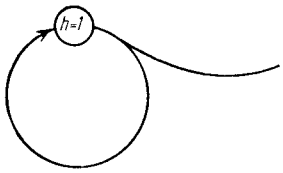
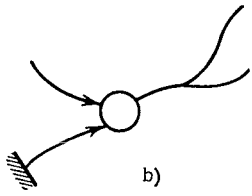


Fig. 4.20.



b)

Fig. 4.21.

of one sampling instant. By connecting q such neurons in series (Fig. 4.23), we can produce a delay of q sampling instants.

Fig. 4.22.

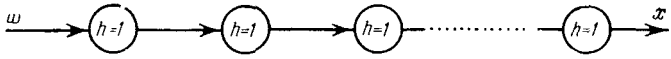


Fig. 4.23.

Consider now a neuron for which $h = 1$, $s = 1$, $q = 1$, and fix the input stimulus (Fig. 4.24). Then the neuron embodies the relationship

$$x^{p+1} = \overline{v^p},$$

that is, the single neuron performs the operation of negation with a delay of one instant.

A neuron for which $h = 1$, $q = 0$, and s is any number (Fig. 4.25) is an embodiment of a disjunction of s variables with a delay of one instant:

$$x^{p+1} = w_1^p \vee w_2^p \vee \ldots \vee w_s^p,$$

while a neuron with $h = s$ (for any s) and $q = 0$ (Fig. 4.26) is an embodiment of a conjunction of s variables with a delay of one instant.

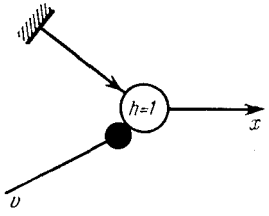


Fig. 4.24.

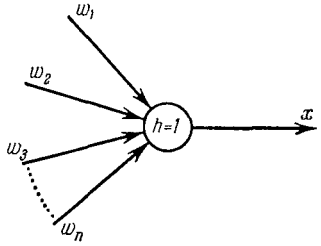


Fig. 4.25.

If we desire to perform a conjunction of s variables, some of which are negated, then the negated variables must be introduced at the inhibitory inputs while h must equal the number of negated variables. Thus, for example, the conjunction

$$x^{p+1} = w_1^p \,\&\, w_2^p \,\&\, \overline{w_3^p} \,\&\, \overline{w_4^p}$$

can be embodied by a single neuron, as shown in Fig. 4.27.

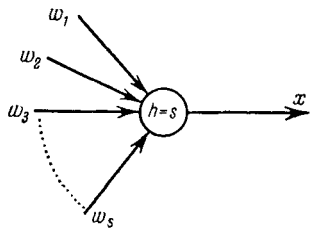


Fig. 4.26.

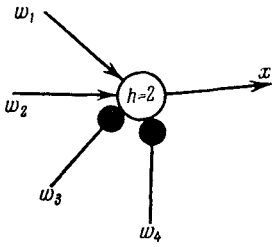


Fig. 4.27.

Consider now an arbitrary disjunction of conjunctive groups. Let such a disjunction contain m groups. Then each conjunctive group may be performed by a single neuron in accordance with Fig. 4.27. The outputs of these neurons are connected to a neuron which performs the disjunction (Fig. 4.25). Since all the neurons performing conjunctions "fire" during one instant, and since just one additional instant is required for performing a disjunction, the entire disjunctive form may be performed in two instants, that is, in time 2τ. Thus,

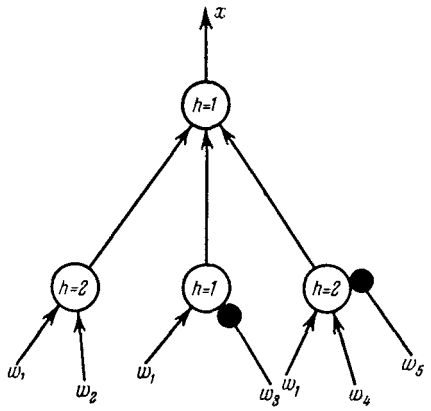


Fig. 4.28.

Fig. 4.28 shows a net consisting of McCulloch-Pitts neurons and embodying the form

$$x^{p+2} = (w_1^p \,\&\, w_2^p) \vee (w_1^p \,\&\, \overline{w_3^p}) \vee (w_4^p \,\&\, \overline{w_5^p} \,\&\, w_1^p).$$

Since any logical function may be presented as a disjunction of conjunctive groups, it can be performed over two sampling instants by an abstract net consisting of McCulloch-Pitts neurons. Thus any

logical converter L may be constructed from McCulloch-Pitts neurons, but, in contrast to our usual assumptions, such a converter will not be instantaneous, because it will require two sampling instants to finish its operation.

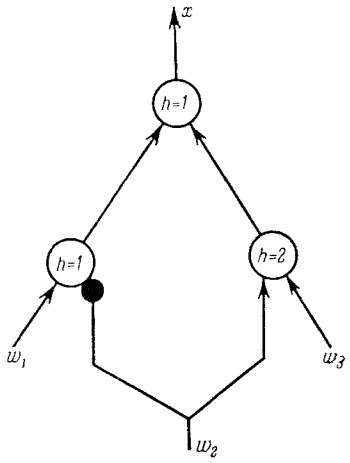


Fig. 4.29.

Assume now that the inputs vary and that the state of the net is observed at instants 0. 2τ, 4τ, ... With such a timing, one can construct a net of McCulloch-Pitts neurons that performs any logical conversion over a single sampling instant. Also, neurons may be employed to form a delay element for such a "doubled" timing; to do this (see Fig. 4.23), two delay elements are connected in series. Now, having a logical converter performing any desired conversion and having a delay element, we can construct a net embodying any desired automaton operating with such a timing.

In Chapter 10 we shall consider methods for synthesizing automata operating with any desired "slow" timing, starting from elements operating with "fast" timing, provided the synchronizing signals for the occurrence of the "slow" timing are supplied from an outside source. We shall show that to synthesize such systems we need elements capable of producing a delay for any such externally supplied timing.

To finish the discussion of neural nets, we shall show how such a delay element may be synthesized from McCulloch-Pitts neurons. Thus let us construct a net (Fig. 4.29) that performs, over time 2τ, the function

$$x^{p+2} = (w_1^p \,\&\, \overline{w_2^p}) \vee (w_3^p \,\&\, w_2^p).$$

We combine two such nets into one net with two inputs (u and u_t), as shown in Fig. 4.30. The input u is the basic input of the net, while u_t is used for introduction of a synchronizing signal (it is assumed that the next sampling instant occurs when u_t changes from 1 to 0).

Figure 4.31 shows the variation of x and x_1 for some variations of u and u_t. The value of x coincides with the value of u, but with a delay of one sampling instant. The net will function correctly provided signals u_t follow each other at intervals not shorter than 4τ. The arrows in Fig. 4.31 bracket intervals 2τ.

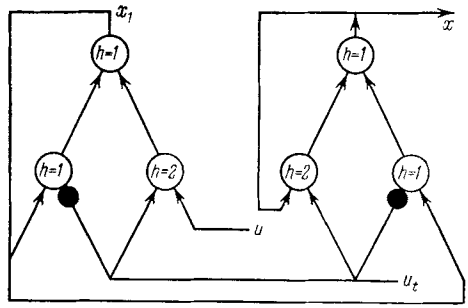


Fig. 4.30.

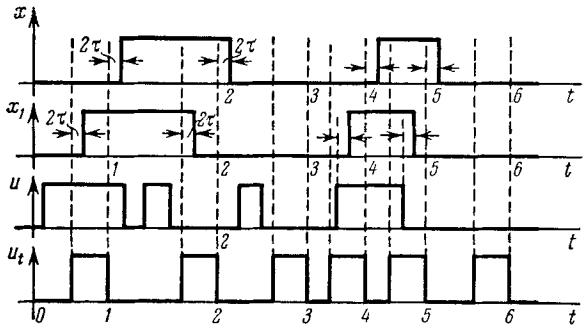


Fig. 4.31.

Thus, having a delay element for any external synchronizing source, as well as a converter "firing" in the time 2τ, one can, with the aid of the methods of Chapter 10, use McCulloch-Pitts neural nets to embody any automaton (or sequential machine) with any desired timing, provided a single condition is satisfied: the interval between sampling instants cannot be shorter than 4τ.

5

Technical Embodiment of Finite Automata and Sequential Machines

5.1. TWO METHODS FOR TECHNICAL REALIZATION OF FINITE AUTOMATA AND SEQUENTIAL MACHINES

In the preceding chapters we have formally introduced the concepts of "finite automaton," "sequential machines," and "abstract structure." So far, these were presented only as equations or systems of equations, and we did not deal with the physical nature of the dynamic systems whose motion they describe. Now we shall show that the above concepts describe important technical systems, and we shall introduce techniques for determining the hardware needed for realizing any given finite automaton or s-machine.

We have shown in Chapter 4 that each finite automaton or s-machine may be represented by many abstract structures. But each abstract structure may be embodied by some practical device that functions just like this abstract structure. It follows that any finite automaton can have many technical embodiments. We shall also show that any given abstract structure of any given automaton may be embodied (realized) by many technical means.

In this chapter we shall consider only embodiments (realizations) of binary abstract structures; that is, it will be assumed that the finite automaton is given by a system of relations

$$x_i^{p+1} = F_i\left[x_1^p,\ x_2^p,\ \ldots,\ x_n^p,\ u_1^p,\ u_2^p,\ \ldots,\ u_s^p\right] \quad (i = 1,\ 2,\ \ldots,\ n), \tag{5.1}$$

where x_i $(i = 1, 2, \ldots, n)$ and u_j $(j = 1, 2, \ldots, s)$ are logical variables which can be only 0 or 1, and F_i $(i = 1, 2, \ldots, n)$ are logical functions, which also can be only 0 or 1. We also assume that the timing of the automaton is given, that is, we are given the conditions defining the occurrence of the discrete moments 0, 1, 2, ..., p on the continuous time scale.

To produce a technical device performing relation (5.1), one must have logical converters performing the functions F_i. We have already described such devices in Chapter 2. Now, however, we do not want to perform functions F_i themselves, but want to embody relations (5.1) of which such functions are a part. Thus, we are faced with the question: What modification must be introduced into the function converters of Chapter 2 (or with what must these converters be supplemented), in order to transform them into devices whose states shall vary in time so as to model the abstract structure (5.1)?

We shall now present two essentially different methods for solving the above problem.

5.2. AGGREGATIVE DESIGN OF FINITE AUTOMATA AND SEQUENTIAL MACHINES

We already know that an abstract structure such as (5.1) can be placed into correspondence with a structural diagram. Such a diagram (for $n = 3$, $s = 2$) is shown in Fig. 5.1. The diagram contains s input lines (input wires $u_1, u_2, \ldots, u_s$) and n output lines (their coordinates are states $x_1, x_2, \ldots, x_n$). Each of the n logical converters performing functions $F_1, F_2, \ldots, F_n$, respectively, receives signals from all the $n + s$ lines; the output of the ith converter feeds the line x_i via a one-instant delay element (denoted by a circle in Fig. 5.1), whose output and input are related by

$$x_{\text{out}}^{p+1} = x_{\text{in}}^{p}.$$

Direct examination shows that such a circuit models precisely the structure of relations (5.1). To construct a technical device according to this diagram, one must have one-instant delay elements in addition to the requisite logical converters. Thus in order to convert a set of elements sufficient for the embodiment of any logical function into a set of elements sufficient for the realization of a finite automaton, one needs only to supplement the first set with a single element—a one-instant delay element.

Such a set is also sufficient for the construction of any sequential machine, since the latter differs from an automaton only in having an output logical converter.

A one-instant delay element must have two inputs—the basic input x_{in} and an auxiliary (time) input x_t. It also must have an output x_{out}. The conventional notation for such an element is shown in Fig. 5.2.* The auxiliary (time) input receives the signals indicating

*One usually omits the input wire x_t whenever such an omission does not hinder the understanding of the operation of the circuit.

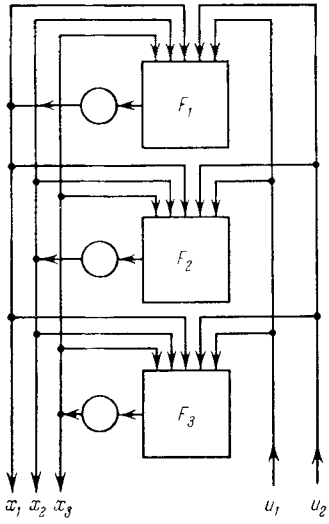


Fig. 5.1.

the occurrence of the next discrete moment, such signals being supplied to the automaton from an external signal-producing device (a "clock" or "synchronous source").

The delay element operates in the following manner: let x^*_{in} be the state of the input to the element at the first discrete moment. Then, after a short

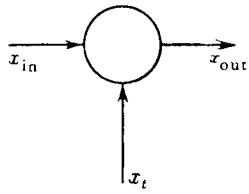


Fig. 5.2.

time interval τ (the specific delay of the delay element), the output shows $x_{out} = x^*_{in}$.

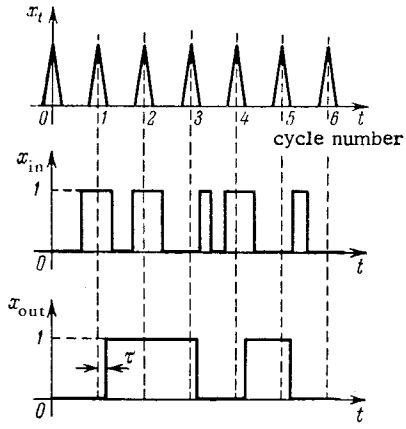


Fig. 5.3.

After this, regardless of what happens at the input, the output will retain its value until the next discrete moment, when the same procedure is repeated. The delay element does not react to any changes occurring at the input during the time interval between the discrete moments.

Figure 5.3 shows an example of changes occurring at the input and output of a one-instant delay element. In this example the synchronizing signals are short pulses. However, the synchronizing signal often is each change of state of the auxiliary input, which can also have only two values: it can be either 1 or 0 (Fig. 5.4) or, alternatively, it can only change from state 0 to state 1 (Fig. 5.5).

Consider the construction of a pneumatic one-instant delay element. Such an element is based on so-called memory cells. Schematic diagrams of two types of memory cell are shown in Figs. 5.6,a and b, respectively, where the change of state of the "time input" P_t from 0 to 1 serves as the synchronizing signal. A memory cell consists of two pneumatic relays (see Section 2.4). One of these (the output) is connected so as to perform a "repetition," maintaining the output pressure P of the cell equal to pressure P_h; the other relay (the input) performs the function of a pneumatic valve, opening or closing the connection between the chamber where the pressure P_h is established and the input line P_1. The operation of the pneumatic valve is governed by the pressure P_t; in the cell of the first type (Fig. 5.6,a) the valve is closed when $P_t = 1$ and open when $P_t = 0$, and, conversely, in the cell of the second type (Fig. 5.6,b) it is closed when $P_t = 0$ and open when $P_t = 1$. Because of this arrangement, either the cell output is equal to its input (for the first cell when $P_t = 0$, and for the second cell when $P_t = 1$), or the output is not connected with the input and is constant (in the first cell when $P_t = 1$, and in the second cell when $P_t = 0$), its value being determined by the magnitude of the pressure P_h in the dead-end chamber.

A memory cell of the first type connected in series with a cell of the second type constitutes a one-instant delay element (Fig. 5.6,c). This element operates in the following way: at t_n, when P_t is 1 (the beginning of the nth discrete moment), the first cell "memorizes" the value of the input, that is, $P^*(t_n) = P_1(t_n)$. In the same instant (more precisely, at time $t_n + \Delta t$, where the increment Δt is caused by the fact that the working membrane of the second memory cell must travel a longer path than that in the first one), the second memory cell transfers the value remembered by the first cell to the output of the system: the pressure $P(t_n) = P^*(t_n) = P_1(t_n)$ is thus established at the output of the delay element. After this, as long as $P_t = 1$, there can be no changes in the system, since its state is determined by the fact that throughout all this time the first cell "remembers" the input value $P_1(t_n)$. This means that $P(t) = P^*(t) = P_1(t_n)$ when $t_n \leqslant t \leqslant t_n'$, where t_n' is the instant at which P_t becomes 0.

At time t_n' (see Fig. 5.6,d) the input to the second memory cell is $P^*(t_n') = P_1(t_n)$; the cell "memorizes" it, and there is thus no

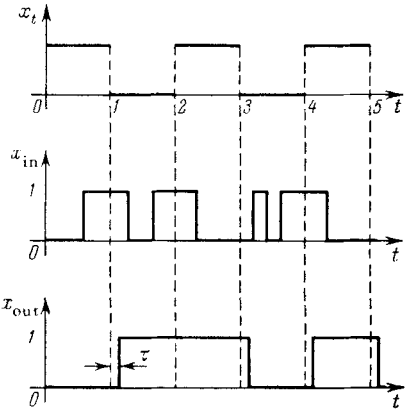


Fig. 5.4.

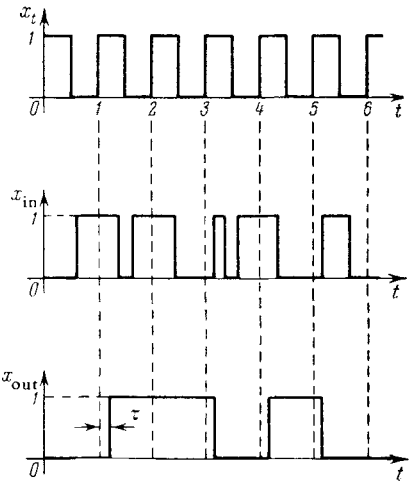


Fig. 5.5.

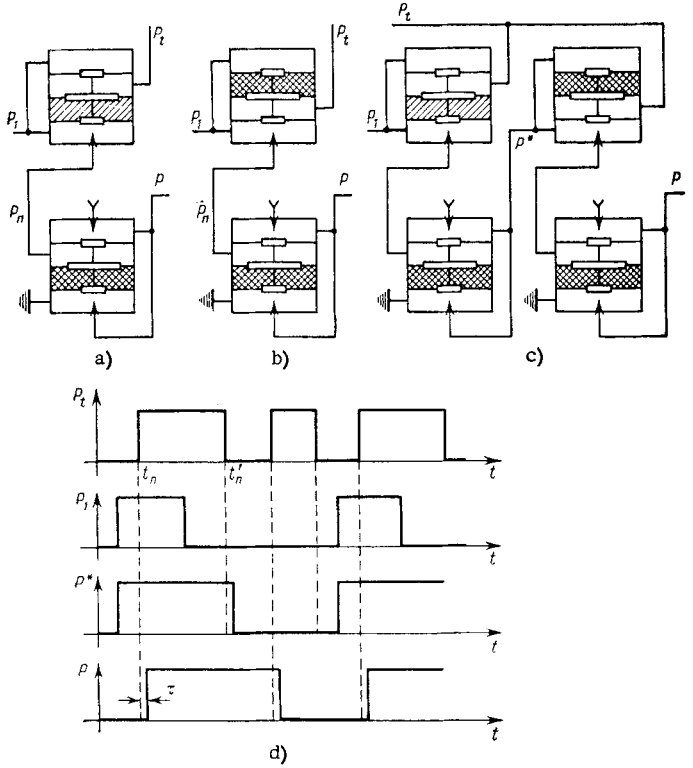


Fig. 5.6.

change in the output of the system, which is still at $P(t'_n) = P_1(t_n)$; at time $t'_n + \Delta t$, the first memory cell starts to operate as a repeater. Subsequently, as long as $P_t = 0$, the output of this system (that is, the delay element) will remain unchanged; it can assume a new value only if, at time t_{n+1}, the control P_t(the time input) becomes 1 again.

Thus this pneumatic device performs the function of a one-instant delay element: its output at the instant of the synchronizing signal (when $P_t = 1$) becomes equal to the input and then, no matter what happens at the input, remains unchanged until the following synchronizing signal (compare Fig. 5.6,d with Fig. 5.5).

Figure 5.7 shows an electromechanical embodiment of a one-instant delay element, which has many conceptual similarities to the above pneumatic delay element. Again, we have two inputs, X and X_t, where X_t is the time input—the change of X_t from 0 to 1 being the synchronizing signal for the delay element. Again the circuit consists of two series-connected memory cells (*1* and *2* in Fig. 5.7,a). The state of the relay coil Y is the output of the element.

The cells memorize by using blocking contacts (contact y^* in cell *1* and contact y in cell *2*). The time input X_t acts on the cell via its associated contacts x_t and $\bar{x}_t$ in such a way that when the first

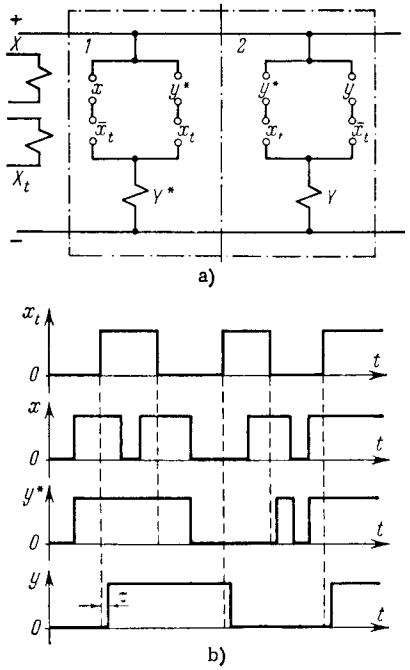


Fig. 5.7.

cell "memorizes" (this will occur at $X_t = 1$), the second cell operates as a repeater of the first one ($Y = Y^*$), and, conversely, when the second cell memorizes (at $X_t = 0$), the first cell repeats the input ($Y^* = X$). Figure 5.7,b shows that this system operates in precisely the same manner as the previously described pneumatic delay element.

As we stated before, a delay element consisting of two memory cells can operate correctly only if the theoretically simultaneous change of state of the cells actually takes place in a certain specified sequence: that is, both cells respond initially by remaining in a state of "memorizing," and only then does one of the cells transform its state into that of a repeater. In a pneumatic element this is achieved by applying differing back pressures P_{h1} and P_{h2}, whereas in the relay switching element this same function is filled by the specific delay τ of relays Y and Y^*.

Any finite automaton may be embodied by replacing the contacts x ($x_1, \ldots, x_s$) of the delay element circuits (such as that of Fig. 5.7) with chains of contacts $f_1, f_2, \ldots, f_n$. Such chains not only incorporate the input contacts $x_1, \ldots, x_s$, but also include the contacts $y_1, \ldots, y_n$ of the output relays of the delay elements. This is shown by the circuit diagram of the automaton (Fig. 5.8). Thus the $u_1, \ldots, u_s$ states of the input fibers of the automaton of Fig. 5.1 correspond to the $x_1, \ldots, x_s$ states of the input contacts of Fig. 5.8 and the $x_1, \ldots, x_n$ states of the automaton of Fig. 5.1 correspond to state of the relay coils $Y_1, \ldots, Y_n$ of Fig. 5.8, and, finally, the logical converters $F_1, \ldots, F_n$ of Fig. 5.1 correspond to the chains of contacts $f_1, \ldots, f_n$ in Fig. 5.8.

Obviously, the one-instant delay element is itself the simplest finite automaton. If one desires to assemble not merely logical converters but also automata then the set of constituent elements must include either a one-instant delay element or some other elementary

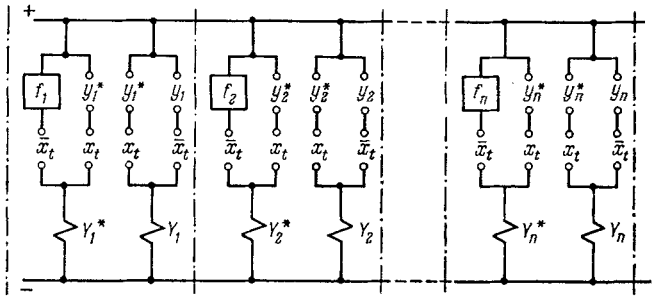


Fig. 5.8.

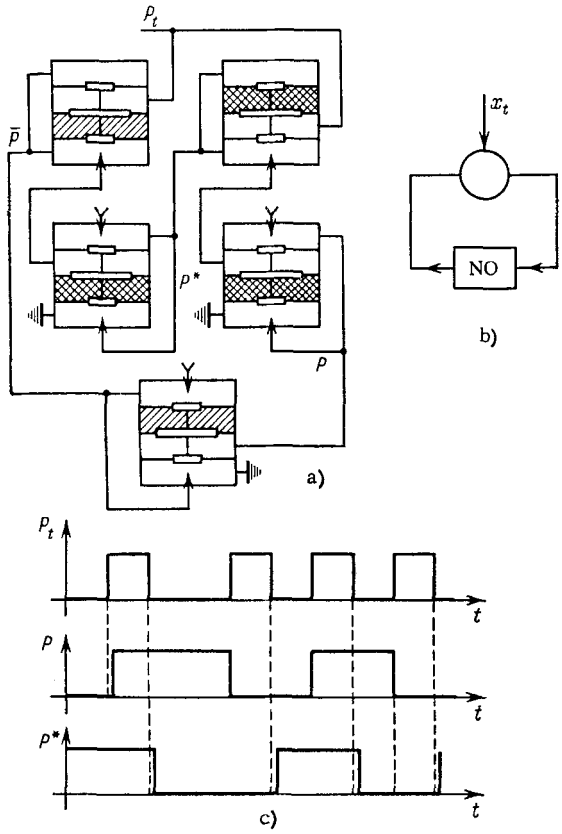


Fig. 5.9.

(nonautonomous) finite automaton. In another widely used method, one supplements the logical elements with an elementary automaton which, although it does not permit construction of all conceivable finite automata, does give many finite automata of practical value. One such elementary automaton is the complementary flip-flop (an autonomous automaton). Figure 5.9 shows a gas-operated flip-flop based on a pneumatic delay element. This flip-flop (Fig. 5.9,a) is obtained from a delay element by switching its output into its own input via a negation element (Fig. 5.9,b). Such a circuit is an autonomous finite automaton operating according to $x^{p+1} = \bar{x}^p$ (here, pressure P substitutes for x), an operation shown in Fig. 5.9,c. Figure 5.10 shows an electromechanical flip-flop, also made from a delay element by switching its output into its own input via a negation element.

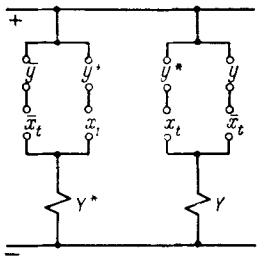


Fig. 5.10.

This technique for the synthesis of automata involves supplementing the set of instantaneously acting logical elements with some very simple automaton (for example, a one-instant delay element, a flip-flop, and so on). In addition, in using delay elements, this technique assumes the availability of a synchronous source whose output becomes the time input of the delay elements. In many cases, however, there is no need for supplementing the logical set with new elements: one merely utilizes the fact that any real element has an inherent time lag τ; that is, any real element is an elementary automaton operating in a discrete time scale devised by dividing the time axis into uniform intervals of length τ. The realization of this fact leads to the most popular (although somewhat limited) technique for synthesizing automata. This technique is applicable when the synchronizing signal, defining the division of the continuous time into discrete moments, is the change of the input state of the system.

5.3. SYNTHESIS OF FINITE AUTOMATA AND SEQUENTIAL MACHINES BY UTILIZING INHERENT DELAYS AS WELL AS FEEDBACK

Consider again the simplest electromechanical relay, which in Chapter 2 was assumed to be acting instantaneously. In fact, however, relay actuation involves a short time lag τ. This means that even though the output (the state of contacts x) and the input (flow of current in coil X), are both logical variables (that is, can only be 0 or 1), their relationship involves a time element. Thus

$$x^{t+\tau} = X^t.$$

If time is uniformly divided into a succession of discrete moments $0, \tau, 2\tau, 3\tau, \ldots$ and if changes of the input as well as all outputs occur only at these moments, we get

$$x^{p+1} = X^p,$$

that is, the relay* is an elementary automaton of the "one-instant time delay" type, operating at intervals τ.

*We are referring here to a relay with normally open contacts. If the actuation time is also taken into account, then a relay with normally closed contacts may be considered as a circuit consisting of a one-instant time delay element and an instantaneous negation element.

Further, a real contact network synthesized by the methods of Chapter 2 will not, in fact, perform the "instantaneous" function

$$x = F(u_1, u_2, \ldots, u_s),$$

but will be an automaton

$$x^{p+1} = F\left[u_1^p, u_2^p, \ldots, u_s^p\right],$$

operating at times $0, \tau, 2\tau, 3\tau, \ldots$.

Consider now a relay network such that normally open contacts of one relay close the circuit of the coil of the succeeding relay (Fig. 5.11). Then the input of the whole network is the current in the coil of the first relay, while its output is the closing of the contact x_n of the last relay. Such a network can be described by

$$x_n^{p+1} = X_n^p = x_{n-1}^p, \quad x_{n-1}^{p+1} = X_{n-1}^p = x_{n-2}^p, \ldots,$$
$$x_2^{p+1} = X_2^p = x_1^p, \quad x_1^{p+1} = X_1^p$$

or

$$x_n^{p+n} = X_1^p,$$

forming a typical loop-free automaton—an n-instants time delay line.

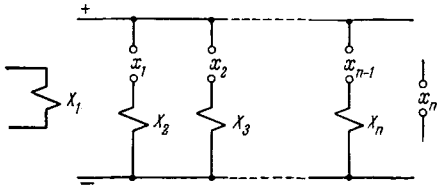


Fig. 5.11.

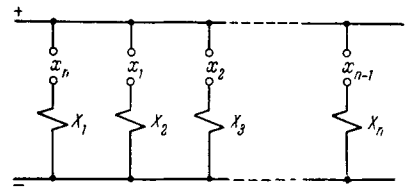


Fig. 5.12.

Let us now construct an automaton with a loop, connecting the coil of the first relay of this delay line with the contacts of the last relay; that is, we shall close the delay circuit by means of feedback (Fig. 5.12). Again, all contacts are normally open. Then, following some initial state of contacts, this automaton, operating at intervals of τ, will assume and stay in one of two possible stable states (all contacts closed or all open). If, however, the first relay was normally closed, while the others were normally open, then we would have continuous cyclic switching of contacts. The diagram of

this automaton would show all its states connected into a closed cycle. In particular, this is how the flip-flop circuit of Fig. 5.13 operates. Considered in this way, any relay switching circuit is an automaton operating at intervals τ. As we have seen, both loop-free automata (for example, the delay line shown in Fig. 5.11) and automata with loops (for example, those of Figs. 5.12 and 5.13) may be synthesized by this method. However, the only automaton of this type which makes sense is the autonomous one, since the assumption that the input also changes at intervals τ would be unrealistic.

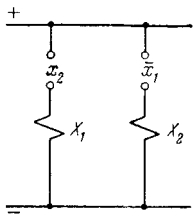


Fig. 5.13.

It should be pointed out that, in the case of loop-free autonomous automata, the diagram can have only one stable point (equilibrium) toward which the automaton tends whatever the initial state. In the case of automata with loops, however (that is, feedback circuits), the diagrams may show closed cycles, several equilibria, and so on (see Chapter 6). Even although such automata are sometimes used, they are not of great value since their cycle timing, that is, the intervals between successive discrete moments, is predetermined by the delay inherent in the relay, and so is usually very fast.

The mostly widely used automata are those in which the cycle timing is governed only by the change of the state of the input, such changes being infrequent and spaced over longer intervals of time than those required for the actuation time τ of the relay. We shall call such a cycle timing *slow*, while the cycle timing in which the time is divided into uniform intervals τ shall be known as *fast*.

Automata with slow timing governed by a change at the input may be synthesized from automata with fast timing, in which case we have a *transformation of timing* (see Chapter 10). To achieve this, one takes advantage of the fact that it is possible to synthesize fast, relay-based autonomous automata whose diagrams show several stable states. Consider, for example, the simplest relay circuit (Fig. 5.14). This circuit contains two relays, whose coils Y_1 and Y_2 are connected in subcircuits which also contain the contacts belonging to these relays. Consequently, we have a feedback circuit or an automaton with loops. In addition, the circuit also includes the contacts x_1 and x_2 of two auxiliary relays X_1 and X_2. These contacts supply the input signals.

Let the input contacts be fixed in some position. Then, if the initial state of the remaining contacts is given, the circuit operates as an autonomous automaton with fast cycle timing, conforming to the diagram of this automaton. If the diagram does not show any closed cycles but has several possible equilibria, the system can

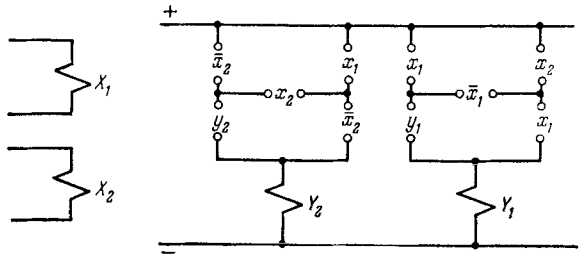


Fig. 5.14.

only tend toward one of these. Precisely which equilibrium state will be attained is determined by the initial state of the system.

Assume that the equilibrium state achieved is A. Then, some time after A has been established, let us change the state of the input contacts; after this, in accordance with previous discussion, the new state of the input contacts remains fixed. Now, with this new state of input contacts, the circuit is a new autonomous automaton with a new diagram. This new diagram may also have several possible equilibria, but the previous equilibrium A need not be one of them. If this is the case, we have a new "transient process"; that is, the automaton begins operating in fast cycle timing, tending toward a new equilibrium B, whose position is governed both by the diagram of the new automaton and by the position of the state A on this diagram.

This process is repeated whenever the state of the input contacts is changed. However, if the input contacts revert to their first state sometime later, the system need not necessarily return to equilibrium A. Indeed, in this case we recreate the initial autonomous automaton with the initial diagrams, but now the point B may be positioned on some branch of the diagram other than that on which the system was initially (prior to establishment of A). The result is that the new state of equilibrium will be other than A; it may, for example, be C, in conformance with our assumption that the diagram of our autonomous automaton shows more than one state of equilibrium.

Let us now imagine that we are recording the states of the inputs and outputs of our relay system α seconds after each change of state of the input contacts. The value of α shall be made so large that all "transient processes" occurring with fast cycle timing will have ended and a state of equilibrium attained by the time the reading is taken. However, α will not be so large that a change in the state of the input will occur during it. Then, at instants α, we shall

observe only equilibrium states; whether some state will occur will depend on the preceding equilibrium state and the state of the inputs; that is, the finite automaton now embodied by the circuit no longer operates with fast cycle timing but with a timing which is governed by the changing of the state of the input.

If the output and perhaps even the input of this automaton, are fed to a logical output converter, we have a sequential machine with slow cycle timing.

The circuit really operates with fast timing, but this is immaterial since we are interested only in the states occurring after the transient processes have terminated, and so we simply neglect these transient processes. We have thus transformed a fast-cycling automaton into a slow-cycling one. This technique of synthesizing finite automata and s-machines is, in reality, the one used for systems based on electromechanical relays, vacuum tubes, semiconductor triodes or diodes.

We come now to the following problem: can all *a priori* defined automata (or s-machines) operating with a timing governed by the change of its input states be synthesized via the above technique? The answer is yes. One method utilizing this technique is described in Section 5.4.

A related problem is that of the most economical network, that is, the network utilizing the above transformation technique to embody a given automaton and, at the same time, containing the least number of relays, contacts and states (or minimizing some other parameters affecting the cost of the circuit). A general solution for one such problem is given in Chapter 10.

The above transformation technique is based on the assumption that the diagrams of the corresponding autonomous automata show several states of equilibrium. However, this is possible only in the case of automata with loops. It follows that a fast automaton must of necessity be one with a loop, which in practice is achieved by means of feedback, that is, by connecting the relay coils to their own contacts. In this sense the resulting networks become slow automata only because of feedback. Relays connected into feedback circuits are sometimes called *intermediate relays*, as distinct from relays that are employed for the control of input contacts *(input relays)* or for picking up the signal resulting in the circuit *(output relays)*.

Comparing the aggregate method of synthesis of automata with that based on multiple stable states, we see that the aggregate method is based on a special element—the one-instant delay element—whereas the technique of multiple stable states requires no

other devices than the very same relays that are used in the logical converters, while the spacing of the operation in time is achieved by means of feedback loops and the special construction of contact networks.

It is quite obvious that all the elements of an aggregate set, in particular, its one-instant time delay elements, can themselves be based on the multiple stable states concept (see the circuit of the relay-based delay element, Fig. 5.7). However, such elements can be utilized in the aggregate systems only in conjunction with output power amplifiers; that is, they must be active.

Relay circuits are frequently designed in such a manner that the diagrams of the autonomous automata, resulting at all possible states of the input contacts, are of the specific form shown in Fig. 5.15; such diagrams show several equilibrium states (where the arrows issuing from these states lead back to the same states), while all the remaining, nonequilibirum, states are directly connected by arrows with equilibrium states. Given such a circuit, only one automaton cycle is required for attaining equilibrium; that is, equilibrium is achieved in time τ. Therefore the time α needs to be only slightly longer than τ. In practice, this means that the state of a slow automaton can be observed almost immediately after a change of the input. It is, of course, understood, that several relays may operate simultaneously during this single cycle.

If the actuating time τ were exactly the same for all the relays, then the fact that several relays are actuated simultaneously would cause no complications. However, in real systems τ is not exactly the same for all relays. For this reason, a system operating with fast cycling time may change states in a sequence different from the one that it would have followed given exactly synchronized relays. In this case, the type of resulting change of state would depend on which relay is the first actuated, that is, on factors that are random and usually not controllable. An example of this phenomenon, known as *critical race* of relays, is given in Section 5.4. This term emphasizes that the operation of the circuit is governed by the relay that operates fastest. Since one should not permit the operation of a circuit to depend on random factors, critical competition of relays must be prevented. To avoid this competition, the circuit embodying a given finite automaton or a sequential machine must satisfy

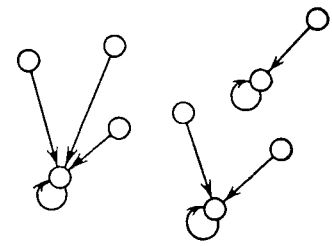


Fig. 5.15.

some additional requirements: for example, one requirement may be that the system shall be transformed from one state to another during a single "fast" cycle via the operation of a single relay. Such additional conditions necessitate more complex circuits, and thus a larger number of constituent elements (relays, contacts). Circuits satisfying these conditions are called *realizations*. There are a number of standard realizations. One of them, proposed by Huffman, will be described in the next section.

Naturally the competition problem does not apply in cases where the relays are strictly synchronized. Such a situation exists with some systems synthesized from magnetic amplifiers and tube elements, since in such systems this time τ is externally imposed on all the elements by the frequency of the alternating current feeding the system.

5.4. HUFFMAN'S METHOD AND REALIZATION

The early Huffman paper [170] on relay switching circuits still does not contain the concept of a sequential machine or a finite automaton, or their equivalents. While citing a number of ways in which the problem of synthesis of a relay switching network may be specified Huffman showed that one method is to start from a special table, which he calls the *flow table*. Assuming thereafter that the flow table is given, Huffman shows how it can be simplified (but does not show the limits of such a simplification), and then develops a general method for synthesis of relay switching circuits embodying this flow table. Huffman's circuit realizes the given table in its equilibrium states. But since his paper was not based on the concepts of a finite automaton and a sequential machine, Hoffman obviously could not specify that his method actually involves an s-machine with fast cycle timing which, in its stable states, realizes the given s-machine. The latter already has slow cycle timing, governed by changes in the states of the input.

We shall now develop Huffman's method, making use of the concepts of finite automaton, sequential machine, and cycle timing transformation. Assume we are given an s-machine, that is, two tables: the base table of the finite automaton involved, and the output converter table. We also assume that the cycle timing of the automaton is governed by change of the states of the input. We want to synthesize a relay network which, in its stable states will realize the given s-machine in accordance with the principles stated in Section 5.3. This problem is solved by Huffman's method on a simple

example, but the method is general and may be applied to other cases in exactly similar manner.

a) The Type of Automaton or Sequential Machine

Recall that any sequential machine (including any finite automaton) may be defined by the system of relations (see Section 3.4)

$$\left.\begin{aligned} \mu &= F(\varkappa, \rho), \\ \varkappa^{p+1} &= \mu^p, \\ \lambda &= \Phi(\varkappa, \rho), \end{aligned}\right\} \tag{5.2}$$

where

$$\mu, \varkappa = \{\varkappa_1, \ldots, \varkappa_k\}, \quad \rho = \{\rho_1, \ldots, \rho_r\}, \quad \lambda = \{\lambda_1, \ldots, \lambda_l\}.$$

This follows from the fact that sequential machines and automata defined by

$$\left.\begin{aligned} \mu &= F(\varkappa, \rho), \\ \varkappa^{p+1} &= \mu^p, \\ \lambda &= \Phi(\mu, \rho), \end{aligned}\right\} \tag{5.3}$$

can always be reduced to the form (5.2) by introducing the function

$$\Phi(\mu, \rho) = \Phi[F(\varkappa, \rho), \rho].$$

The converse is not true; that is, a system defined in the form (5.2), can only occasionally be reduced to (5.3) without a change in the number of states. In particular, an automaton defined by

$$\left.\begin{aligned} \mu &= F(\varkappa, \rho), \\ \varkappa^{p+1} &= \mu^p, \\ \lambda &= \mu, \end{aligned}\right\} \tag{5.4}$$

can always be represented by

$$\left.\begin{aligned} \mu &= F(\varkappa, \rho), \\ \varkappa^{p+1} &= \mu^p, \\ \lambda &= F(\varkappa, \rho), \end{aligned}\right\} \tag{5.5}$$

whereas an automaton given by

$$\left.\begin{aligned} \mu &= F(\varkappa, \rho), \\ \varkappa^{p+1} &= \mu^{p}, \\ \lambda &= \varkappa, \end{aligned}\right\} \tag{5.6}$$

cannot be reduced to form (5.3).

In Section 3.4, s-machines defined by (5.2) were called P -P machines, and s-machines given by (5.3) were named P - Pr machines. We shall again use this terminology here.

Since the equation system (5.2) is universal, it can be used as a canonical method for defining finite automata and sequential machines. System (5.2) finds correspondence in two tables, each with the arguments $\varkappa$ and ρ. In order to emphasize that the arguments are common to both tables, and also for the sake of conciseness, we shall treat these two tables as one. Thus the data required for the synthesis of the relay circuit will be presented as a combined table of the automaton and the output converter. The columns of this table correspond to various input states, and the rows to the various states of the automaton; in accordance with (5.2), the squares of this table shall contain two symbols: that denoting the state of the automaton, and that describing the state of the output of the s-machine. The combined table may be interpreted in the following manner. It may be assumed that the headings of the rows and columns correspond to the current states of the automaton and the input (the states at time p); then the symbols in the squares define the next state of the automaton (state at time $p + 1$) and the current state of the output of the s-machine. This is in accord with the representation of system (5.2) in the form

$$\begin{aligned} \varkappa^{p+1} &= F(\varkappa^{p}, \rho^{p}), \\ \lambda^{F} &= \Phi(\varkappa^{p}, \rho^{p}). \end{aligned}$$

Tables 5.1 - 5.5 show how the starting data may be given by means of combined tables.

Table 5.1 defines an automaton that may be interpreted as being either of the P - P or the P - Pr type (there is a one-to-one correspondence between λ and $\varkappa$ in all the squares of the table).

Table 5.2 gives an automaton that cannot be of the P - Pr type (different symbols λ may correspond to the same symbol $\varkappa$ within a single column; for example, we have pairs $x_3\lambda_2$ and $x_3\lambda_3$ in column ρ_1).

Table 5.1

$\varkappa$ \ ρ	ρ_1	ρ_2	ρ_3	ρ_4
$\varkappa_1$	$\varkappa_3\lambda_2$	$\varkappa_4\lambda_4$	$\varkappa_3\lambda_2$	$\varkappa_2\lambda_1$
$\varkappa_2$	$\varkappa_3\lambda_2$	$\varkappa_4\lambda_4$	$\varkappa_1\lambda_3$	$\varkappa_4\lambda_4$
$\varkappa_3$	$\varkappa_2\lambda_1$	$\varkappa_4\lambda_4$	$\varkappa_2\lambda_1$	$\varkappa_4\lambda_4$
$\varkappa_4$	$\varkappa_1\lambda_3$	$\varkappa_1\lambda_3$	$\varkappa_3\lambda_2$	$\varkappa_2\lambda_1$

Table 5.2

$\varkappa$ \ ρ	ρ_1	ρ_2	ρ_3	ρ_4
$\varkappa_1$	$\varkappa_3\lambda_2$	$\varkappa_4\lambda_2$	$\varkappa_3\lambda_2$	$\varkappa_2\lambda_2$
$\varkappa_2$	$\varkappa_3\lambda_3$	$\varkappa_4\lambda_3$	$\varkappa_1\lambda_3$	$\varkappa_4\lambda_3$
$\varkappa_3$	$\varkappa_2\lambda_4$	$\varkappa_4\lambda_4$	$\varkappa_2\lambda_4$	$\varkappa_4\lambda_4$
$\varkappa_4$	$\varkappa_1\lambda_1$	$\varkappa_1\lambda_1$	$\varkappa_3\lambda_1$	$\varkappa_2\lambda_1$

Table 5.3 defines a sequential machine that can be reduced to a P - Pr machine (there is a unique relationship between each $\varkappa$ and λ within each column; in this particular case there is also a one-to-one correspondence between $\varkappa$ and λ in each column).

Table 5.3

$\varkappa$ \ ρ	ρ_1	ρ_2	ρ_3	ρ_4
$\varkappa_1$	$\varkappa_3\lambda_1$	$\varkappa_4\lambda_2$	$\varkappa_3\lambda_2$	$\varkappa_2\lambda_4$
$\varkappa_2$	$\varkappa_3\lambda_1$	$\varkappa_4\lambda_2$	$\varkappa_1\lambda_4$	$\varkappa_4\lambda_3$
$\varkappa_3$	$\varkappa_2\lambda_3$	$\varkappa_4\lambda_2$	$\varkappa_2\lambda_1$	$\varkappa_4\lambda_3$
$\varkappa_4$	$\varkappa_1\lambda_2$	$\varkappa_1\lambda_1$	$\varkappa_3\lambda_2$	$\varkappa_2\lambda_4$

Table 5.4

$\varkappa$ \ ρ	ρ_1	ρ_2	ρ_3	ρ_4
$\varkappa_1$	$\varkappa_3\lambda_1$	$\varkappa_4\lambda_2$	$\varkappa_3\lambda_2$	$\varkappa_2\lambda_4$
$\varkappa_2$	$\varkappa_3\lambda_1$	$\varkappa_4\lambda_2$	$\varkappa_1\lambda_4$	$\varkappa_4\lambda_3$
$\varkappa_3$	$\varkappa_2\lambda_3$	$\varkappa_4\lambda_3$	$\varkappa_2\lambda_1$	$\varkappa_4\lambda_3$
$\varkappa_4$	$\varkappa_1\lambda_2$	$\varkappa_1\lambda_1$	$\varkappa_3\lambda_2$	$\varkappa_2\lambda_4$
a_i	3	3	3	2

Tables 5.4 and 5.5 define sequential machines that cannot be reduced to the P - Pr type. Table 5.4 differs from Table 5.5 in that in each of its columns there is a unique relationship between λ and $\varkappa$; that is, each λ is always paired with the same $\varkappa$ but there is no overall one-to-one correspondence between $\varkappa$ and λ.

Tables 5.1 - 5.4 have one common property: in each of them, the next state of the automaton (symbol $\varkappa$ in a square) is uniquely

Table 5.5

$\varkappa$ \ ρ	ρ_1	ρ_2	ρ_3	ρ_4
$\varkappa_1$	$\varkappa_3\lambda_1$	$\varkappa_4\lambda_2$	$\varkappa_3\lambda_2$	$\varkappa_2\lambda_4$
$\varkappa_2$	$\varkappa_3\lambda_1$	$\varkappa_4\lambda_2$	$\varkappa_1\lambda_4$	$\varkappa_4\lambda_3$
$\varkappa_3$	$\varkappa_2\lambda_3$	$\varkappa_4\lambda_1$	$\varkappa_2\lambda_1$	$\varkappa_4\lambda_3$
$\varkappa_4$	$\varkappa_1\lambda_2$	$\varkappa_1\lambda_1$	$\varkappa_3\lambda_2$	$\varkappa_2\lambda_4$

defined by the current states of the input and output of the s-machine. Table 5.5 does not follow this rule.

Huffman's method may be used directly where the given circuit is not of the form of Table 5.5. In other words, this method realizes all automata (both those that can and cannot be reduced to the P - Pr type), all sequential machines that can be reduced to the P - Pr type, and some sequential machines that cannot be reduced to the P - Pr type but have the properties specified in Table 5.4.

b) Development of a Flow Table from a Given Table for an s-Machine

It is required to develop a sequential machine defined by its stable states. The machine is given, in a P - P form, by a *basic table*, which is the combined table of the automaton and the converter (Table 5.6). We assume that the next (discrete) sampling instant, that is, the cycle timing of this machine, is governed by the instant of change of its input states. We also assume that the next state of the automaton is uniquely defined by the current states of the input and output of the s-machine, that is, that there are no table columns where one λ is paired with different $\varkappa$. Table 5.6 satisfies this requirement. To solve this problem by Huffman's method, we must convert this table into a *flow table*. Do this as follows:

Table 5.6

$\varkappa$ \ ρ	ρ_1	ρ_2	ρ_3	ρ_4
$\varkappa_1$	$\varkappa_2\lambda_1$	$\varkappa_2\lambda_1$	$\varkappa_1\lambda_3$	$\varkappa_1\lambda_1$
$\varkappa_2$	$\varkappa_1\lambda_3$	$\varkappa_3\lambda_2$	$\varkappa_3\lambda_1$	$\varkappa_1\lambda_1$
$\varkappa_3$	$\varkappa_1\lambda_2$	$\varkappa_2\lambda_1$	$\varkappa_1\lambda_3$	$\varkappa_2\lambda_2$
α	3	2	2	2

We add to machine Table 5.6 a bottom row where we enter α_i, the number of different output states (different λ) the machine can have at any given ith input (in our case $\alpha_1 = 3$, $\alpha_2 = \alpha_3 = \alpha_4 = 2$. We then compute $\sigma = \sum_{i=1}^{r} \alpha_i$, where r is the number of different input states. In our case $r = 4$ and $\sigma = \alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 = 9$. After this, we develop Table 5.7 which has the same

number of columns as the basic Table 5.6, but has σ rows. The input (top) row again contains ρ, while the extreme left column carries a sequence of new variables $x' = \{x'_1, \ldots, x'_\sigma\}$; in our case $x' = \{x'_1, \ldots, x'_9\}$. Now we fill out the table. We copy into column ρ_1 (beginning with its first row) the various λ (total number α_1) from column ρ_1 of the basic table (here these are λ_1, λ_3, and λ_2). The sequence in which these are entered into Table 5.7 is immaterial. Next to these symbols λ we copy the symbols x' denoting the given rows of Table 5.7. These squares correspond to equilibrium states; let us mark them with rectangles. We fill in the α_2 squares of column ρ_2 in the same manner, but here the entries do not start from the first row but from the first blank row (here, row x'_4). We thus enter pairs $x'_4\lambda_1$ and $x'_5\lambda_2$. In a similar manner we complete the α_i squares of each ith column.

Now, we find the symbol λ in the top row of column ρ_1 of Table 5.7 (in our case, this is λ_1); we find the square containing the same

Table 5.7

x' \ ρ	ρ_1	ρ_2	ρ_3	ρ_4	x''
x'_1	$\boxed{x'_1}\lambda_1$	$x'_5\lambda_2$	$x'_7\lambda_1$	$x'_9\lambda_1$	x''_1
x'_2	$\boxed{x'_2}\lambda_3$	$x'_4\lambda_1$	$x'_6\lambda_3$	$x'_9\lambda_1$	x''_2
x'_3	$\boxed{x'_3}\lambda_2$	$x'_4\lambda_1$	$x'_6\lambda_3$	$x'_9\lambda_1$	x''_3
x'_4	$x'_2\lambda_3$	$\boxed{x'_4}\lambda_1$	$x'_7\lambda_1$	$x'_9\lambda_1$	x''_4
x'_5	$x'_3\lambda_2$	$\boxed{x'_5}\lambda_2$	$x'_6\lambda_3$	$x'_8\lambda_2$	x''_5
x'_6	$x'_1\lambda_1$	$x'_4\lambda_1$	$\boxed{x'_6}\lambda_3$	$x'_9\lambda_1$	x''_6
x'_7	$x'_3\lambda_2$	$x'_4\lambda_1$	$\boxed{x'_7}\lambda_1$	$x'_8\lambda_2$	x''_7
x'_8	$x'_2\lambda_3$	$x'_5\lambda_2$	$x'_7\lambda_1$	$\boxed{x'_8}\lambda_2$	x''_8
x'_9	$x'_1\lambda_1$	$x'_4\lambda_1$	$x'_6\lambda_3$	$\boxed{x'_9}\lambda_1$	x''_6

symbol in the first column of Table 5.6 and note the $\varkappa$ appearing in that pair (in our case this is $\varkappa_2$).* We find the row headed by this $\varkappa$ in the basic table and, retaining the same sequence, copy the symbols λ from this row into the corresponding blank squares of the first row of Table 5.7 (here, λ_2, λ_1 and λ_1, respectively, are copied below ρ_2, ρ_3 and ρ_4). We proceed in the same manner with all rows of Table 5.7, and we enter symbols λ in all its squares. Now we match each lone λ of Table 5.7 with a symbol $\varkappa'$ in such a manner that in each column the λ's are uniquely paired with $\varkappa'$; in other words, we pair the λ's of each column, with the symbols $\varkappa'$ enclosed by the equilibrium state rectangles. Now we have a full Table 5.7, which is equivalent to Huffman's flow table, and represents the table of a "fast" sequential machine defined in the P - P form. If the ρ and λ variables are sampled only in equilibrium states, this "fast" machine will operate exactly as the starting one (Table 5.6).

c) Abbreviation (Compression) of the Flow Table

If the flow table has one or more identical rows, then this automaton may be replaced by another "fast" automaton with a smaller number of states. We shall illustrate this with Table 5.7.

We introduce the new variable $\varkappa''$, and place it into correspondence with $\varkappa'$, making sure that wherever there are one or more identical rows in Table 5.7, they correspond to the same $\varkappa''$. This matching is best done by adding a right-hand side column of $\varkappa''$ to the table of the first "fast" automaton. In our particular case, this column (see Table 5.6) has two identical $\varkappa''$ values (in the sixth and ninth rows), corresponding to identical rows. We then develop the compressed flow Table 5.8; it contains the same number of columns as Table 5.7, but only as many rows as there are different symbols $\varkappa''$, and $\varkappa''$ replaces $\varkappa'$ throughout. The compressed table thus obtained defines (in the P - P form) a "fast machine," which, when sampled only at its states of equilibrium, operates in the same way as the starting "slow" machine, but has a minimum number of states (this last statement will be clarified in Chapter 10). This, in essence, concludes the synthesis of the sequential machine. All that remains is to realize the machine by means of a relay circuit.

---

*In view of the restrictions imposed on the basic table, there is always one matching $\varkappa$ for this λ, although the same $\varkappa$ may appear in more than one row of each column.

Table 5.8

x'' \ ρ	ρ_1	ρ_2	ρ_3	ρ_4
x_1''	$\boxed{x_1''}\lambda_1$	$x_5''\lambda_2$	$x_7''\lambda_1$	$x_6''\lambda_1$
x_2''	$\boxed{x_2''}\lambda_3$	$x_4''\lambda_1$	$x_6''\lambda_3$	$x_6''\lambda_1$
x_3''	$\boxed{x_3''}\lambda_2$	$x_4''\lambda_1$	$x_6''\lambda_3$	$x_6''\lambda_1$
x_4''	$x_2''\lambda_3$	$\boxed{x_4''}\lambda_1$	$x_7''\lambda_1$	$x_6''\lambda_1$
x_5''	$x_3''\lambda_2$	$\boxed{x_5''}\lambda_2$	$x_6''\lambda_3$	$x_8''\lambda_2$
x_6''	$x_1''\lambda_1$	$x_4''\lambda_1$	$\boxed{x_6''}\lambda_3$	$\boxed{x_6''}\lambda_1$
x_7''	$x_3''\lambda_2$	$x_4''\lambda_1$	$\boxed{x_7''}\lambda_1$	$x_8''\lambda_2$
x_8''	$x_2''\lambda_3$	$x_5''\lambda_2$	$x_7''\lambda_1$	$\boxed{x_8''}\lambda_2$

d) Compilation of the Table of the Relay Network

We encode the states of the "fast" automaton in binary notation (Table 5.9), and do the same for its inputs ρ (Table 5.10) and outputs λ (Table 5.11).

We now introduce relays in the same number as there are digits in the binary coding of a given state. We use two digits to code the inputs ρ (Table 5.10) and thus have two input relays. We also have two output relays, from coding of Table 5.11. The states of the automaton (Table 5.9) are matched to intermediate relays, of which there are thus three.

We then rewrite Table 5.8 (utilizing the notations of Tables 5.9 - 5.11) to obtain Table 5.12 (the automaton table) and Table 5.13, the output converter table.

Now we have logical functions which can be realized by actual circuits: the codes employed as headings of the rows and the columns of Tables 5.12 and 5.13 give the values of logical variables

Table 5.9

α_1''	000
α_2''	001
α_3''	010
α_4''	011
α_5''	100
α_6''	101
α_7''	110
α_8''	111

(the states of the contacts of input and intermediate relays), whereas the digits in the matrices themselves are the values of the logical functions (currents in the coils of the intermediate and output relays).

To develop an expanded table of logical functions for our example, let us use x_1 and x_2 for the

Table 5.10

ρ_1	00
ρ_2	01
ρ_3	10
ρ_4	11

Table 5.11

λ_1	00
λ_2	01
λ_3	10
λ_4	11

the states of the contacts of the input relays, y_1, y_2, and y_3 for the states of the contacts of the intermediate relays, Y_1, Y_2, and Y_3 for the states of the coils of the intermediate relays, and Z_1 and Z_2 for the states of the coils of the output relays. We now obtain Table 5.14.

Table 5.12

States of contacts of the intermediate relays	States of contacts of the input relays			
	x_2x_1 0 0	x_2x_1 0 1	x_2x_1 1 0	x_2x_1 1 1
0 0 0	0 0 0	1 0 0	1 1 0	1 0 1
0 0 1	0 0 1	0 1 1	1 0 1	1 0 1
0 1 0	0 1 0	0 1 1	1 0 1	1 0 1
0 1 1	0 0 1	0 1 1	1 1 0	1 0 1
1 0 0	0 1 0	1 0 0	1 0 1	1 1 1
1 0 1	0 0 0	0 1 1	1 0 1	1 0 1
1 1 0	0 1 0	0 1 1	1 1 0	1 1 1
1 1 1	0 0 1	1 0 0	1 1 0	1 1 1
$y_3y_2y_1$	$Y_3Y_2Y_1$	$Y_3Y_2Y_1$	$Y_3Y_2Y_1$	$Y_3Y_2Y_1$

The top part of Table 5.14 shows all the possible combinations of states of contacts x_1, x_2, y_1, y_2, and y_3. For each column of this table, there is an entry in Tables 5.12 and 5.13. We then copy into each of the rows Y_1, Y_2, Y_3, Z_1, and Z_2 of the lower part of Table 5.14 the

Table 5.13

States of contacts of the intermediate relays	States of contacts of the input relays			
	x_2x_1 0 0	x_2x_1 0 1	x_2x_1 1 0	x_2x_1 1 1
0 0 0	0 0	0 1	0 0	0 0
0 0 1	1 0	0 0	1 0	0 0
0 1 0	0 1	0 0	1 0	0 0
0 1 1	1 0	0 0	0 0	0 0
1 0 0	0 1	0 1	1 0	0 1
1 0 1	0 0	0 0	1 0	0 0
1 1 0	0 1	0 0	0 0	0 1
1 1 1	1 0	0 1	0 0	0 1
$y_3y_2y_1$	Z_2Z_1	Z_2Z_1	Z_2Z_1	Z_2Z_1

numbers contained in the corresponding positions of Tables 5.12 and 5.13, and we thus complete Table 5.14.

Once we have this table of logical functions, we can use any desired method to derive the circuit corresponding to it (see Section 2.3).

e) Huffman's Realization of the Circuit

So far we have attempted to design a circuit that would substitute for a given sequential machine but we have not required a realization, that is, we did not ensure hazard-free operation. For example, we did not prevent simultaneous actuation of several relays. Thus, in the example of the preceding section there are conditions under which transition from one equilibrium to another will be accompanied by simultaneous actuation of several relays. For example, assume that the automaton is in state $Y_1 = 0$, $Y_2 = 1$, $Y_3 = 0$, with inputs at $x_1 = 0, x_2 = 0$ (first column, third row, Table 5.12). Then at a new input $x_1 = 1$, $x_2 = 1$ it will go to state $Y_1 = 1$, $Y_2 = 0$, $Y_3 = 1$. This transition is accompanied by simultaneous actuation of all three relays Y_1, Y_2 and Y_3. If some of the relays are faster than others, for example, if Y_1 and Y_3 are already set to 1 when Y_2 is still in the process of being set to 0, the circuit may assume a stable state $Y_1 = 1$, $Y_2 = 1$, $Y_3 = 1$; that is, the automaton may work in an undesirable fashion because its relays have inherent delays or operate in an improper sequence.

An automaton based on delay elements gives completely hazard-free operation. It is, however, rather difficult to design the automaton from delay elements when it is required to operate with a

Table 5.14

y_1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
y_2	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
y_3	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
x_1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
x_2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Y_1	0	1	0	1	0	0	0	1	0	1	1	1	0	1	1	0	0	1	1	0	1	1	0	0	1	1	1	1	1	1	1	1
Y_2	0	0	1	0	1	0	1	0	0	1	1	1	0	1	1	0	1	0	0	1	0	0	1	1	0	0	0	0	1	0	1	1
Y_3	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Z_1	0	0	1	0	1	0	1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1
Z_2	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0

cycle timing governed by the change of the input. In this case, we would also need a synchronous source which would respond to all change at the input, in order to provide a timing signal for the delay elements.

However, the flow table is actually the basic table of a "fast" automaton which corresponds to the initial automaton in the sense that sampling of its stable states gives a pattern describing the operation of that initial automaton (which works in a timing governed by change of the input). We can therefore design from it a "fast" automaton, corresponding to the initial one, using delay elements, and thus ensure hazard-free operation. This is easier to accomplish, since in this case the construction of the synchronous timing source is simpler.

The Huffman realization is, in reality, such a procedure. We design a "fast" automaton network from the flow table, using delay elements based on relays, and we organize a relay switching synchronous source. Thus the circuit of Huffman's realization contains an automaton based on delay elements (see Fig. 5.8) with contact networks f_i defined by the flow table.

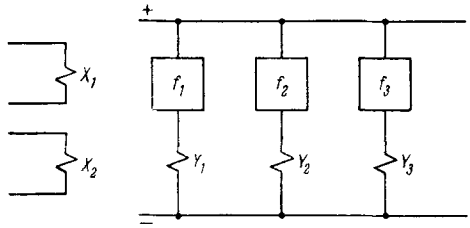


Fig. 5.16.

If the flow table contains m rows, then this part of the circuit will contain $2s_0$ relays (two relays in each delay element), where the number s_0 of delay elements is equal to the smallest integer satisfying the inequality $m \leqslant 2^{s_0}$.

The contact networks f_i are defined by the same logical functions of the flow table that define the states of the intermediate relays Y_i during the design of the switching network while ignoring hazards.

Figures 5.16 and 5.17 show block diagrams of relay switching networks corresponding to the automaton synthesized in the example of the preceding paragraph. The circuit of Fig. 5.16 ignores the possibility of hazards whereas that of Fig. 5.17 is a Huffman hazard-free realization. In these diagrams f_1, f_2, and f_3 are the contact networks (same for both tables) defined by Table 5.14. These contact networks may be synthesized from Table 5.14 by any desired method (for example, Bloch's method described in Section 2.3).

The contacts x_t and $\bar{x}_t$ of Fig. 5.17 govern the delay elements and belong to a special relay X_t at the output of the synchronous source (clock). Huffman has presented a generalized circuit for such a source. It is based on the following considerations.

The flow table is so constructed that after a change of its input and at the end of one "fast" cycle, the corresponding automaton is in equilibrium; that is, its state remains stable during subsequent

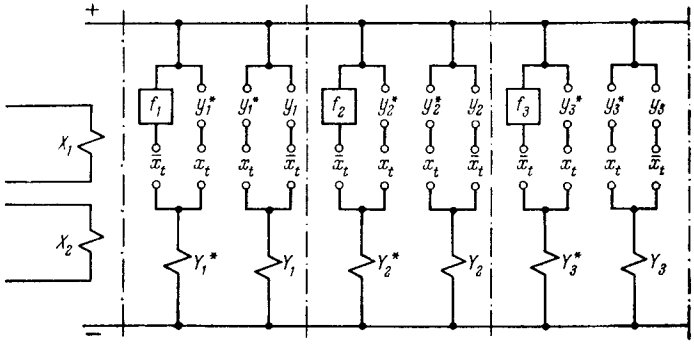


Fig. 5.17.

"fast" cycles. If delay elements are used in the circuit, the state of equilibrium means the equivalence of inputs and outputs in each delay element. Thus the beginning of a new cycle may be associated with the instant at which there is an inequality (nonequivalence) between the input and output in some delay element (that is, when for any delay element $Y_i^* \nabla Y_i = 1$), provided that such a nonequivalence results from a change at the input and occurs in all those delay elements where it should occur. This condition, with the additional restriction that the synchronous source must return to its initial state only when equivalence between inputs and outputs has been

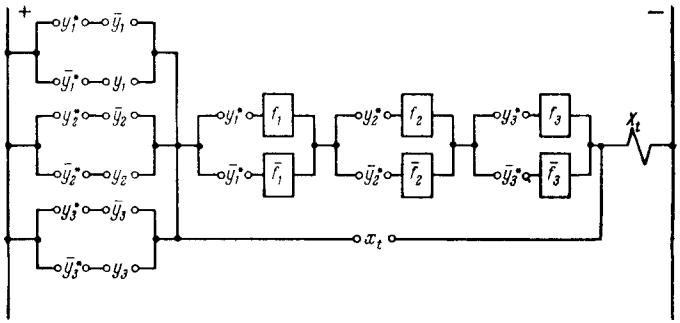


Fig. 5.18.

reestablished in the delay elements, leads to the following logic for the clock:

$$X_t = [(y_1^* \nabla y_1) \vee (y_2^* \nabla y_2) \vee \ldots \\ \ldots \vee (y_n^* \nabla y_n)] \,\&\, \{[(y_1^* \sim f_1) \,\&\, (y_2^* \sim f_2) \,\&\, (y_n^* \sim f_n)] \vee x_t\}.$$

A relay switching circuit of such a clock at $n = 3$ (to correspond with our example), is shown in Fig. 5.18. Since the clock adds one more relay to the $2s_0$ relays already employed in the delay elements, this Huffman realization is called the $(2s_0 + 1)$ realization.

6

Autonomous Finite Automata and Sequential Machines

6.1. WHAT AUTONOMOUS FINITE AUTOMATA AND SEQUENTIAL MACHINES "CAN DO"

This and the following chapter will deal with the capabilities of finite automata and sequential machines. We shall establish what they "can do" and shall determine what is in principle beyond the "capabilities" of any such machine. In the present chapter we shall determine the capabilities of the autonomous finite automaton.

Recall that a finite automaton is *autonomous* if the variable ρ under function F is fixed. Recall also that from this automaton, whose equation is

$$\varkappa^{p+1} = F(\varkappa^p, \rho^p),$$

one can compile r different autonomous automata (see Section 3.7)

$$\varkappa^{p+1} = F(\varkappa^p, \rho_*), \tag{6.1}$$

because ρ_* can coincide with any of the r symbols of alphabet $\{\rho\}$. The constant ρ_* may be treated as a symbol-parameter.

If $\varkappa^0$ is given, then by virtue of (6.1) we can find $\varkappa^1$. Then, substituting $\varkappa^1$ into (6.1), we can determine $\varkappa^2$, and so on. Thus, an autonomous finite automaton started at some initial symbol $\varkappa^0$ can generate an infinite sequence of symbols $\varkappa$:

$$\varkappa^0, \varkappa^1, \varkappa^2, \ldots, \varkappa^p, \ldots$$

The number of symbols in alphabet $\{\varkappa\}$ is finite and equal to k. For this reason, the automaton will generate, after a finite number of time units q $(q \leqslant k)$, a symbol that has already appeared in the sequence, for example, the symbol $\varkappa_6$ in the sequence

$$\varkappa_2\varkappa_3\varkappa_6\varkappa_5\varkappa_4\varkappa_7\varkappa_6 \ldots$$

Thus, from the $(q+1)$-th time unit onward, the automaton will simply periodically repeat this symbol sequence (the sequence of our example is $\varkappa_2\varkappa_3\underline{\varkappa_6\varkappa_5\varkappa_4\varkappa_7}\varkappa_6\varkappa_5\varkappa_4\varkappa_7\varkappa_6\varkappa_5\varkappa_4\varkappa_7 \ldots$ where the recurring sequence is underlined).

It follows that an autonomous automaton, started up at any initial state will, after at most k time units, periodically repeat a sequence of symbols $\varkappa$ (whose length does not exceed k). With any other initial state, this automaton would also periodically repeat a sequence of symbols $\varkappa$ after some time. However, this sequence may not coincide with that generated before.

A special case occurs when the sequence consists of a simple symbol. This will occur when, for example,

$$\varkappa_2\varkappa_3\varkappa_6\varkappa_6\varkappa_6 \ldots$$

or

$$\varkappa_2\varkappa_2\varkappa_2\varkappa_2 \ldots$$

A symbol that appears twice in succession in a sequence shall be called a *stable* symbol. We shall say that the automaton becomes stable during the time unit in which the stable symbol first appears.

Symbol $\varkappa_j$ is stable if

$$\varkappa_j = F(\varkappa_j, \rho_*), \tag{6.2}$$

that is, at $\rho = \rho_*$, symbol $\varkappa_j$ is translated by Eq. (6.1) into itself.

All these statements are illustrated by the diagram of the autonomous automaton. This diagram contains k circles, corresponding to all the possible symbols $\varkappa$ that can be generated. Several arrows may converge on each circle, but only one may be drawn from it. For this reason, if we proceed in the direction of the arrows, we are bound to return after no more than k steps to one of the circles previously crossed (the arrows form a *cycle*, Fig. 6.1), or, alternatively, we shall arrive to a circle at which the leaving arrow forms a loop, that is, indicates equilibrium (see Fig. 6.2).

Figure 6.3 shows various diagrams for $k = 12$.

In the case of Fig. 6.3, the cycle involves all the available circles. Here the machine generates from the outset a periodically

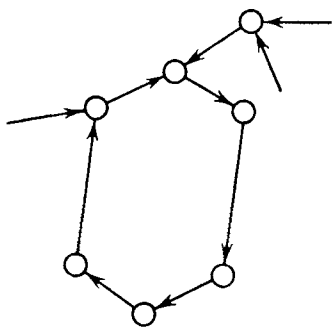


Fig. 6.1.

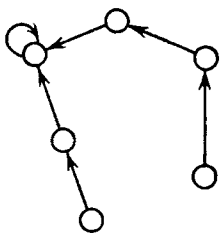


Fig. 6.2.

recurring sequence consisting of all k symbols.* The initial state determines only the order of the symbols in the sequence.

In Fig. 6.3,b the cycle encompasses only five circles; the arrows drawn from the remaining circles converge on one of the circles of the cycle. Here the automaton generates a sequence of length 5; this sequence will be generated after a maximum of one time unit, regardless of the initial state of the automaton.

For the case of Fig. 6.3,c the maximum number of time units preceding the generation of the periodic sequence is four.

Figure 6.3,d shows an autonomous automaton that, depending on its initial state, generates one of three sequences (of lengths 2, 3, and 5, respectively).

The automaton of Fig. 6.3,e can, depending on its initial state, either attain equilibrium after a maximum of three time units, or

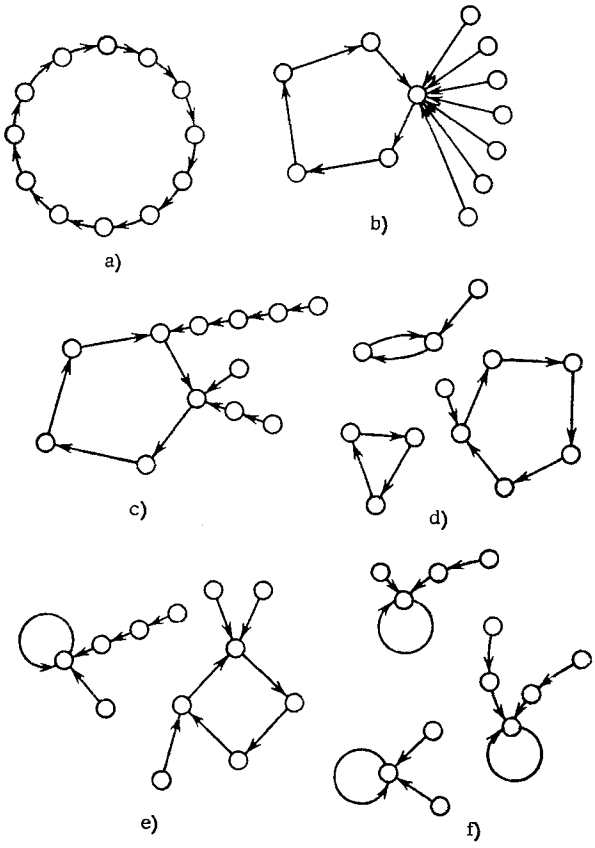


Fig. 6.3.

*In this and the following case we are assuming that it is immaterial what the initial symbol of the sequence is.

can start to generate periodic sequence of length 4 after a maximum of one time unit.

Finally, Fig. 6.3,f shows an autonomous automaton with three equilibrium states: depending on its initial state, some equilibrium is attained in at most two discrete time units.

If we want to synthesize an autonomous automaton generating a periodic sequence of some given length q and we impose some additional conditions as to the time preceding the generation of this sequence, all we need to do is to draw a diagram satisfying these conditions. This is because the diagram defines the autonomous automaton uniquely [the basic table can be directly reconstructed from it (Chapter 3)].

Suppose that we want to generate periodic sequences of lengths 2, 4, and 6. Which of the sequences shall be generated shall be determined by the initial state of the automaton; in no case can the generation of the sequence begin later than one time unit after start up.

The minimum k satisfying these conditions is 12. If $k = 12$ (Fig. 6.4), then the appropriate diagram is drawn by joining into cycles any of two, four, and six circles. If, however, $k > 12$, for example, if $k = 16$ (Fig. 6.5), then, to satisfy the conditions, the arrows emerging from circles not included in the cycles are directed to any of the circles of the three cycles.

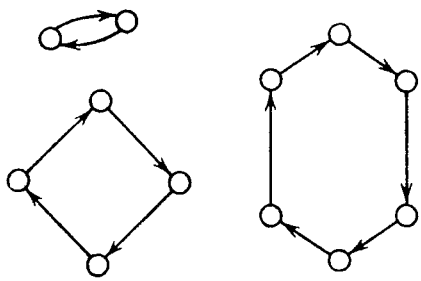


Fig. 6.4.

Problems involving variation of the parameter ρ_* are solved in a similar manner. For each $\rho = \rho_*$ there is an autonomous automaton, that is, a diagram. However, all these automata operate in the same alphabet $\{\varkappa\}$. For this reason, the diagrams of all autonomous automata obtained by varying ρ_* contain the same number k of circles (nodes). Thus assume as before that the automaton generates periodic sequences of lengths 2, 4, and 6. But now it will be the parameter ρ_* (ρ_1, ρ_2 or ρ_3) which will determine which of these sequences shall be generated. As before, the generation of the sequence

at any ρ_* shall begin no later than one discrete time unit after start up of the automaton.

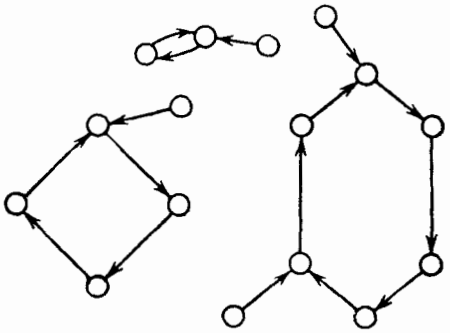


Fig. 6.5.

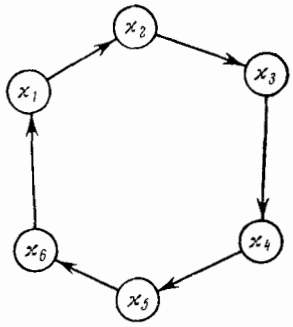


Fig. 6.6.

Obviously, k must be at least equal to the length of the longest given sequence; otherwise it would be impossible to design an autonomous automaton generating that sequence at any ρ. Now, assuming, for example, that $k = 6$, we construct a diagram for $\rho_* = \rho_3$ (Fig. 6.6), connecting all the six nodes into a cycle. Then the graph for $\rho_* = \rho_2$ also contains six nodes, but only four of them (any four) need to be connected into a cycle. The arrows drawn from the remaining two circles are therefore made to converge on the nodes of the cycle (Fig. 6.7).

Finally, only two nodes are connected into a cycle for $\rho_* = \rho_1$, while the arrows from the remaining circles are made to converge on these nodes (Fig. 6.8).

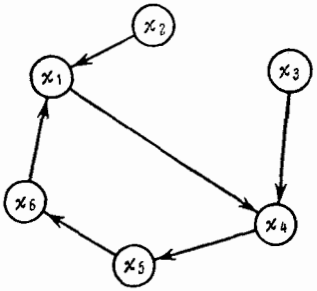


Fig. 6.7.

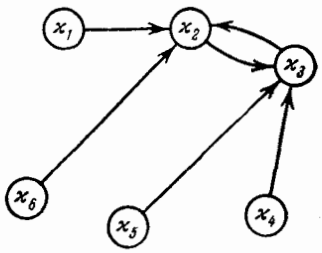


Fig. 6.8.

These three diagrams define the automaton, and its basic table can be readily derived from them (Table 6.1).

Now let us proceed to the sequential machine, where the automaton is supplemented by an output converter.* If the automaton periodically generates a sequence of symbols $\varkappa$, then the converter also must periodically generate a sequence of symbols λ of the same length. The sequence of $\varkappa$ cannot contain two identical symbols, but the output sequence of λ may do so. For example, if the automaton generates the recurrent sequence $\varkappa_1\varkappa_7\varkappa_2\varkappa_5\varkappa_3$, while the converter embodies table

$\varkappa$	$\varkappa_1$	$\varkappa_2$	$\varkappa_3$	$\varkappa_5$	$\varkappa_7$
λ	λ_1	λ_1	λ_2	λ_2	λ_1

,

the sequence $\lambda_1\lambda_1\lambda_2\lambda_2\lambda_1$ will periodically recur at the output of the converter.

Table 6.1

$\varkappa$ \ ρ_*	ρ	ρ_1	ρ_3
$\varkappa_1$	$\varkappa_2$	$\varkappa_4$	$\varkappa_2$
$\varkappa_2$	$\varkappa_3$	$\varkappa_1$	$\varkappa_3$
$\varkappa_3$	$\varkappa_2$	$\varkappa_4$	$\varkappa_4$
$\varkappa_4$	$\varkappa_3$	$\varkappa_5$	$\varkappa_5$
$\varkappa_5$	$\varkappa_3$	$\varkappa_6$	$\varkappa_6$
$\varkappa_6$	$\varkappa_2$	$\varkappa_1$	$\varkappa_1$

If the desired sequence of λ is given, then one synthesizes an s-machine producing this sequence by first synthesizing an automaton generating any sequence of $\varkappa$ of the same length as the given sequence of λ. Then, by writing these two sequences one below the other, we have produced the converter table.

The above discussion has important practical implications.

If a practical device can be considered described by the abstraction "autonomous finite automaton," and if the symbols $\varkappa$ characterize the possible states of this device, then the device can either attain equilibrium in a finite time, or it can periodically repeat a finite sequence of states. It can do nothing else. It also follows that given any finite number of sequences of states of any finite length, we can always synthesize an automaton generating any of these sequences by a judicious choice of its initial state. Alternatively, we can synthesize this automaton by identifying the parameter which can remain constant throughout the operation and fixing its value at the very outset.

To show the practical significance of this technically important conclusion we shall now discuss the synthesis of a bistable abstract structure substituting for an autonomous sequential machine, after which we shall generalize our reasoning to other abstract structures.

---

*Since ϱ_* is fixed, in an autonomous automaton, "sequential machine" and "finite (autonomous) automaton with an output converter" mean the same thing.

6.2. SYNTHESIS OF THE BISTABLE STRUCTURE OF AN AUTONOMOUS SEQUENTIAL MACHINE

Consider first the case when parameter $\rho = \rho_*$ cannot be varied; that is, we do not wish to synthesize a one-parameter family of finite automata or sequential machines, but just one such machine. We formulate the problem as follows: given the number of initial states which determines the number of possible variants of the operation of the machine; for each of these states we are given a sequence of 0 and 1 that the machine must generate, starting the generation no later than one time unit after the beginning of the operation. It is desired to synthesize the bistable structure of the finite automaton and the bistable (logical) converter satisfying the given conditions. We must determine not only the logical functions in the right-hand sides of the equations for the bistable abstract structure, but also the number n of such equations, whereby it is required to obtain a solution with the minimum n. We shall consider the problem solved if we can synthesize a bistable abstract structure with a minimum number of equations, and shall forego further simplification of these equations by means of propositional calculus to meet optimization criteria.

This problem may be divided into for subproblems:

1. Determination of the minimal number n.
2. Synthesis of a finite automaton whose output consists of sequences of the given length.
3. Synthesis of a bistable abstract structure which can substitute for this finite automaton.
4. Synthesis of the required output converter.

Consider first the case when we are given only one sequence of 0 and 1 of length q; it is required to generate it periodically (assuming that the first output symbol can be any of these symbols) from any initial state, the s-machine producing this sequence beginning with the second discrete time unit after the start of its operation.

Let us select the smallest n satisfying the inequality

$$2^n \geqslant q$$

and consider an automaton having $k = 2^n$ states. We shall make the alphabet of its states $\{\varkappa\}$ to coincide with the series of integers $\{0, 1, 2, \ldots, 2^n - 1\}$. Assume that the diagram of this automaton shows the first q nodes connected into a cycle; if $2^n > q$, then each of the remaining nodes is directly connected (by an arrow) to some node of

the cycle. For example, let us connect these "extra" nodes with the node denoted by 0 (Fig. 6.9 shows an example for $n = 3$, $q = 5$).

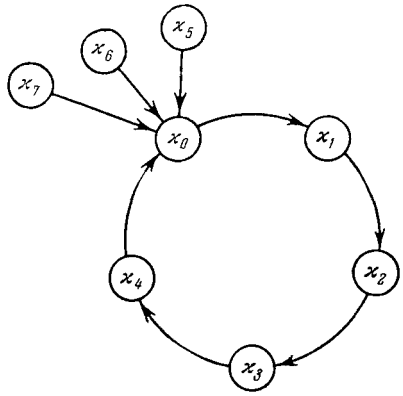


Fig. 6.9.

Table 6.2

$\varkappa$	p	p_*
$\varkappa_0$		$\varkappa_1$
$\varkappa_1$		$\varkappa_2$
$\varkappa_2$		$\varkappa_3$
$\varkappa_3$		$\varkappa_4$
$\varkappa_4$		$\varkappa_0$
$\varkappa_5$		$\varkappa_0$
$\varkappa_6$		$\varkappa_0$
$\varkappa_7$		$\varkappa_0$

This diagram immediately gives the basic table of the autonomous automaton. Thus, we obtain Table 6.2 for Fig. 6.9.

Now let us synthesize the bistable abstract structure of this automaton. We do this by the method of Section 4.2, by selecting n logical coordinates $x_1, x_2, \ldots, x_n$ (each of which may assume values only from the alphabet $\{0, 1\}$), and completing Table 6.3. The table contains 2^n rows and is divided in two parts by columns headed x^p and x^{p+1}.

Column $\varkappa^p$ is filled from top to bottom with numbers 0, 1, 2, ..., $2^n - 1$, that is, the subscripts of $\varkappa$ in the first column of the basic Table 6.2, while column $\varkappa^{p+1}$ contains the subscripts $\varkappa$ from the second column of that basic table.

Now we fill out the left-hand part of Table 6.3 (columns x^p) with binary representation of numbers contained in column $\varkappa^p$. It is convenient to use the rule already described in Chapter 1. Thus the last column on the right (for x_0) is filled with alternating 0 and 1. Column x_1 has alternating pairs of 0 and 1, column x_2— alternating quartets of 0 and 1, and so on. Thus the number of 0 and 1 in the alternating groups doubles in each succeeding column from the right.

We also fill the right part of the table (column x^{p+1}) with binary representation of numbers contained in column $\varkappa^{p+1}$.

Figure 6.4 shows the completion of the generalized Table 6.3 for the particular automaton of Fig. 6.9 and Table 6.2.

Now we turn to the x_0 column on the right of Table 6.3 (or 6.4), and underline all rows where $x_0 = 1$. We write out a conjunction of all x_k contained in the first such row, putting a negative sign over those x which are 0 in the same row of the left part of the table. In the same way, we write out conjunctions for the remaining underlined rows.

Table 6.3

x^p					χ^p	χ^{p+1}	x^{p+1}				
x_{n-1}	...	x_2	x_1	x_0			x_{n-1}	...	x_2	x_1	x_0
0	...	0	0	0	0	1	0	...	0	0	1
0	...	0	0	1	1	2	0	...	0	1	0
0	...	0	1	0	2	3	0	...	0	1	1
0	...	0	1	1	3	4	0	...	1	0	0
0	...	1	0	0	4	0	0	...	0	0	0
0	...	1	0	1	5	0	0	...	0	0	0
0	...	1	1	0	6	0	0	...	0	0	0
0	...	1	1	1	7	0	0	...	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...
1	...	1	1	1	2^n-1	...	...	...	...	...	...

Table 6.4

Value of x^p			χ^p	χ^{p+1}	Value of x^{p+1}		
x_2	x_1	x_0			x_2	x_1	x_0
0	0	0	0	1	0	0	1
0	0	1	1	2	0	1	0
0	1	0	2	3	0	1	1
0	1	1	3	4	1	0	0
1	0	0	4	0	0	0	0
1	0	1	5	0	0	0	0
1	1	0	6	0	0	0	0
1	1	1	7	0	0	0	0

The conjunctive groups thus obtained are then joined by disjunction symbols. The full disjunctive form of the conjunctive groups thus obtained is the right-hand part of the first of the desired relations of the binary abstract structure. For example, for Table 6.4 this relationship is

$$x_0^{p+1} = [\overline{x_0^p} \,\&\, \overline{x_1^p} \,\&\, \overline{x_2^p}] \vee [\overline{x_0^p} \,\&\, x_1^p \,\&\, \overline{x_2^p}] \equiv \overline{x_0^p} \,\&\, \overline{x_2^p}. \qquad (6.3)$$

We shall now treat the x_1 column on the right of Table 6.3 (or 6.4) in an analogous manner and generate the right-hand part of the

second required relation. For Table 6.4, this becomes

$$x_1^{p+1} = [x_0^p \& \overline{x_1^p} \& \overline{x_2^p}] \vee [\overline{x_0^p} \& x_1^p \& \overline{x_2^p}] \equiv [x_0^p \sim \overline{x_1^p}] \& \overline{x_2^p}. \tag{6.4}$$

For column x_2, we obtain

$$x_2^{p+1} = x_0^p \& x_1^p \& \overline{\overline{x_2^p}} \tag{6.5}$$

and so on, until all the n rows have been examined and all the required relations of the desired binary abstract structure have been generated.

Now we synthesize the output converter

$$y = L(x_0, x_1, \ldots, x_{n-1}) \tag{6.6}$$

in the following manner. The q nodes of the cycle in the diagram of our autonomous automaton (see Fig. 6.9) are already numbered consecutively from 0 to $q - 1$. We assign the same numbers (from 0 to $q - 1$) to the 0 and 1 symbols comprising our given sequence (whose length is q). Thus, for example, if our given sequence has a length of 5 and the form 10010, we create the following one-to-one correspondence:

1 0 0 1 0
0 1 2 3 4

Let us separate the nodes which correspond to the symbol 1 of the given sequence. In our example, these nodes are 0 and 3, and they correspond to states $\varkappa_0$ and $\varkappa_3$.

We now compile a conjunction characterizing the values of all the coordinates of one of the states we separated, for example, $\varkappa_*$. To do this we compile a conjunction of all the x_i and place a negation sign over those which are 0 in the row $\varkappa_*$ in the left-hand part of Table 6.3. We form similar conjunctions for all the states that we have separated, and we join these conjunctions by means of disjunction symbols. In our particular example, we separated states $\varkappa_0$ and $\varkappa_3$. The row for $\varkappa_0$ in Table 6.4 contains only zeros, and thus we put a negation sign over all three coordinates x_0, x_1 and x_2. To obtain the conjunction

$$\overline{x_0} \& \overline{x_1} \& \overline{x_2}.$$

State $\varkappa_3$ yields (from the fourth row of Table 6.4) the conjunction

$$x_0 \& x_1 \& x_2.$$

Joining these conjunctions, we get the logical function

$$L(x_0, x_1, x_2) = (\overline{x_0} \& \overline{x_1} \& \overline{x_2}) \vee (x_0 \& x_1 \& \overline{x_2}). \tag{6.7}$$

It is immediately seen that this logical function equals 1 if and only if state x_i coincides with any of the above-separated states. that is, when 1 should be the output of the converter. Therefore, this expression $L(x_0, x_1, \ldots, x_{n-1})$ does indeed define the desired functional converter.

(These disjunctive descriptions of the operation of the automaton and the converter may, of course, be further simplified by the methods of propositional calculus to meet any criterion of optimality.)

The above binary abstract structure of the autonomous s-machine is a solution of our problem, and contains the necessary and sufficient number of states n. This number is sufficient because we were able to synthesize the required automaton with this value of n. It is necessary because at a smaller n the number of nodes in the diagram would be smaller than q, and therefore even if all the nodes were connected into a cycle, the number of time units in the period would be less than q; in that case, only a sequence smaller than q could be generated.

Now let us solve another problem. Assume we are given m different sequences of length q_i $(i = 1, 2, \ldots, m)$. Among them there may also be sequences of length 1. We select the smallest n satisfying the inequality

$$2^n \geqslant \sum_{i=1}^{i=m} q_i. \tag{6.8}$$

If

$$2^n = \sum_{i=1}^{i=m} q_i,$$

then the diagram of the automaton can have precisely m cycles, each of length q_i $(i = 1, 2, \ldots, m)$. If, however,

$$2^n > \sum_{i=1}^{i=m} q_i,$$

the diagram will contain nodes not connected into the cycles, and we shall draw arrows from each of them to one of the nodes of the cycle. After this, the bistable abstract structure of the automaton and the output converter are synthesized in precisely the same manner as in the case when one sequence was generated. In synthesizing the

converter we write out consecutively all the given m sequences and we number consecutively all the symbols (that is, 0 and 1) contained in these sequences.

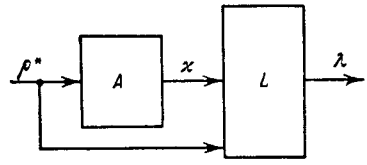


Fig. 6.10.

Now let us consider the case when the controlling parometers of the sequential machine ρ_*, can be varied, that is, when the binary structure

$$x_i^{p+1} = L_i[x_0^p, x_1^p, \ldots, x_{n-1}^p; u_1^p, u_2^p, \ldots, u_s^p], \quad i = 0, 1, 2, \ldots, n-1, \tag{6.9}$$

contains s binary parameters allowing 2^s possibilities.

Assume we are given m sequences. It is required to construct a binary abstract structure of an s-machine in accordance with Fig. 6.10, that is, we require a binary structure of the automaton A [Eq. (6.9)] and of the converter

$$y = L[x_0, x_1, \ldots, x_{n-1}; u_1, u_2, \ldots, u_s] \tag{6.10}$$

and we want the machine to generate the given m sequences. The additional requirement is that n and s should be minimum. The sequence to be generated is selected by choosing one of the parameters $u_1, u_2, \ldots, u_s$.

Let us separate the longest of the m given sequences. Let its length be $q_{\max}$, that is, let it contain $q_{\max}$ symbols. We find the minimum n and s satisfying the inequalities

$$2^n \geqslant q_{\max}, \tag{6.11}$$

$$2^s \geqslant m. \tag{6.12}$$

Let us call these values of n and s respectively $n_{\min}$ and $s_{\min}$. We now draw m diagrams, each of which contains $2^{n_{\min}}$ nodes. Then we number our given sequences consecutively. In the first diagram, we connect into a circle as many nodes as there are symbols in the first sequence, and draw arrows from the "extra" nodes to some node of the cycle. We do the same in the second diagram, except that now the cycle contains as many nodes as there are symbols in the second sequence, and so on. There will be a diagram for all our m sequences because, by virtue of (6.11), the number of nodes in each diagram is at least as large as the number of symbols in any given sequence.

Now we use the previously described method to develop a table such as 6.3 for each diagram; that is, we determine the right-hand

parts in relations

$$x_i^{p+1} = L_i\left[x_0^p, x_1^p, \ldots, x_{n_{\min}}^p\right], \quad i = 0, 1, \ldots, n_{\min} - 1, \qquad (6.13)$$

for each diagram of the binary abstract structure.

Assume that for the jth diagram ($j = 1, 2, \ldots, m$) these relations are

$$x_i^{p+1} = L_{ij}\left[x_0^p, x_1^p, \ldots, x_{n_{\min}}^p\right], \quad i = 0, 1, \ldots, n_{\min} - 1, \qquad (6.14)$$

and that m relations are defined in this manner.

Now we introduce $s_{\min}$ binary variables u_k ($k = 1, 2, \ldots, s_{\min}$) and compile a table of all possible combinations of u_k. This table (Table 6.5) is developed in the same manner as the left-hand part of Table 6.3. Table 6.5 contains $2^{s_{\min}}$ rows. The extreme right-hand column contains numbers of the first m rows [this is possible since by virtue of (6.12) the number of rows is not smaller than m].

Table 6.5

u_k					Number of u
$u_{s_{\min}-1}$	$u_{s_{\min}-2}$	...	u_1	u_0	
0	0	...	0	0	1
0	0	...	0	1	2
0	0	...	1	0	3
...	...	...	...	...	...
...	...	...	...	...	m
...	...	...	...	...	...
1	1	...	1	1	$2^{s_{\min}}$

Returning to relation (6.14) for the jth diagram, we add to the right-hand parts of all the $n_{\min}$ relations involved, conjunctions of those symbols u_k for which we have 1 in the jth row of Table 6.5. Thus we replace (61.4) with a relation

$$x_i^{p+1} = L_{ij}\left[x_0^p, x_1^p, \ldots, x_{n_{\min}-1}^p\right] \& u_{\alpha_1} \& u_{\alpha_2} \& \ldots \& u_{\alpha_k}, \quad i = 0, 1, 2, \ldots, n_{\min} - 1. \qquad (6.15)$$

By consecutively assuming $j = 1, 2, \ldots, m$, we can write m systems of relations such as (6.15).

We now combine by a disjunction the right-hand sides of all the first relations obtained from these m systems, then proceed likewise with respect to the right-hand sides of the second relations in these systems, and so on. As a result we get a single system of relations

$$\left.\begin{aligned} x_i^{p+1} = {} & \{L_{i1}[x_0^p, x_1^p, \ldots, x_{n_{\min}-1}^p] \,\&\, \mathfrak{U}_1\} \vee \\ & \vee \{L_{i2}[x_0^p, x_1^p, \ldots, x_{n_{\min}-1}^p] \,\&\, \mathfrak{U}_2\} \vee \\ & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ & \vee \{L_{im}[x_0^p, x_1^p, \ldots, x_{n_{\min}-1}^p] \,\&\, \mathfrak{U}_m\}, \\ & i = 0, 1, 2, \ldots, n_{\min} - 1, \end{aligned}\right\} \tag{6.16}$$

where $\mathfrak{U}_j$ $(j = 1, 2, \ldots, m)$ denotes the conjunction of all the terms u_k which are equal to 1 in the jth row of Table 6.5.

Let us treat all the u_k as parameters and let us select them, for example, in accordance with the third row of Table 6.5. Then $\mathfrak{U}_3 = 1$, and all remaining $\mathfrak{U}$ are zero; this means that only the third conjunctive term will remain in each of relations (6.16). Equation system (6.16) then coincides with system (6.14) for $j = 3$. The diagram of this system consists of a cycle containing as many nodes as there are symbols (that is, 0 and 1) in the third of our given sequences.

Relations (6.16) are therefore an abstract structure substituting for a finite automaton from which, by selecting vector $\mathfrak{U}$ in accordance with the first m rows of Table 6.5, one can regenerate m diagrams of the same type as above, ensuring the generation of m sequences of given lengths.

In order for these sequences to coincide with the given ones, it remains to synthesize the output converter. To do this we first synthesize (by the methods described above) an output converter for each system of relations (6.14) separately, so that the jth relation (6.14) can generate the jth of the given m sequences.

Let these output converters be

$$y = L_j(x_1, \ldots, x_{n_{\min}-1}). \tag{6.17}$$

Then the desired converter to be added to (6.16) is specified by the relation

$$\begin{aligned} y = {} & [L_1(x_0, x_1, \ldots, x_{n_{\min}-1}) \,\&\, \mathfrak{U}_1] \vee \\ & \vee [L_1(x_0, x_1, \ldots, x_{n_{\min}-1}) \,\&\, \mathfrak{U}_2] \vee \ldots \\ & \ldots \vee [L_m(x_0, x_1, \ldots, x_{n_{\min}-1}) \,\&\, \mathfrak{U}_m]. \end{aligned}$$

It can readily be seen that this synthetic concept and procedure hold not only for a binary abstract structure, but also for any other structure.

In every case the abstract structure must yield a diagram in which there are as many modes in a cycle as there are characters in the given sequence. The transition from diagrams to the structure is accomplished by means of a table such as 6.3, but the completion of such a table and the number of its rows are determined by the particular properties of the synthesized structure, that is, the alphabet of its symbols.

7

Representation of Events in Finite Automata and Sequential Machines

7.1. STATEMENT OF THE PROBLEM

Chapter 3 introduced the concept of tapes of a finite automaton (ρ, $\varkappa$ tape, Table 7.1) and of a sequential machine (ρ, $\varkappa$, λ-tape, Table 7.2). Thus, we know that a tape represents the operation of a finite automaton (or an s-machine) when the input sequence of ρ's and the initial state $\varkappa^0$ are given. The sequences of ρ from 0 time to time p is finite; but since there is no limitation on the operating time of the automaton, the length of each sequence, even though finite, may be as long as desired, and the number of possible sequences of ρ is

Table 7.1

Discrete moment	0	1	2	3	4	5	...	p	...
ρ							...		...
$\varkappa$									

Table 7.2

Discrete moment	0	1	2	3	4	5	...	p	...
ρ							...		...
$\varkappa$							...		...
λ									

infinite. The automaton (or s-machine), which starts from an initial state $\varkappa^0$, establishes a correspondence between each sequence of ρ and some sequence of $\varkappa$ (or λ, in the case of an s-machine); that is, it transforms a sequence of symbols of one alphabet into a sequence of symbols from another alphabet.

What then are the general rules governing this transformation?

Let K be the set of all possible sequences of $\varkappa$, and E be the set of all possible sequences of ρ. The two sets are equipollent. This means that each sequence from K can be placed in correspondence with a sequence from E. Now, if such a correspondence is established in some *arbitrary* way, is it possible to devise a finite automaton embodying this correspondence? Alternatively, is it possible to indicate those correspondences between sequences that can be embodied in a finite automaton, and those that cannot? If there are correspondences that cannot be embodied in any finite automaton, can these be separated from those that can? Identical problems arise with the sequential machines.

These problems can also be formulated in other terms. Assume a finite automaton with a fixed initial state $\varkappa^0$, and consider some state $\varkappa^*$. In examining the ρ, $\varkappa$ tape of the automaton, we mark all instances where $\varkappa^*$ appears. Then we write out all the sequences of ρ (beginning with discrete time 0) that lead to the generation of $\varkappa^*$. Assume that we could analogously process all the conceivable $\rho, \varkappa$ tapes of the same automaton. Assume also that in a set E of all conceivable input sequences of some automaton, we can distinguish a subset G^* of all input sequences that lead to the generation of $\varkappa^*$ in our first automaton. We shall then say that the automaton with initial state $\varkappa^0$ *represents* the input sequences of subset G^* by producing the symbol $\varkappa^*$ at the output. Similarly, an s-machine *represents* input sequences of a subset G^* by generating the symbol λ^* at the output. Our problem then is: Can any subset of input sequences be represented in an automaton or s-machine? If not, what are the conditions for representability of a set of input sequences? What are the properties of representable sets?

To answer these questions we shall first have to formulate the problem more precisely. Therefore, we shall introduce the term "event," and define the classification of events.

7.2. EVENTS. REPRESENTATION OF EVENTS

Let us examine the top strip, that is, the ρ sequence of the tape of an automaton (or a sequential machine). Let us call this strip the *input tape* of the machine (example: Table 7.3).

Table 7.3

Discrete moment	0	1	2	3	4	5	6	7	...
ρ	ρ_3	ρ_1	ρ_5	ρ_7	ρ_1	ρ_{12}	ρ_1	ρ_3	...

Let G be the set of all conceivable input tapes of a given automaton. Further, let us agree that there is some criterion for distinguishing subset G^* from the set G. Whatever this criterion, we shall say that *event G^* occurs* whenever (that is, at all the p's such that) the input tape of the automaton from time 0 to time p belongs to subset G^*.

With time, that is, as p increases, the tape may cease to belong to subset G^*; that is, an unfolding input tape may belong to G^* at some values of p and not belong to G^* at other p's.

Example 1. The event occurs if ρ_5 and ρ_3 are consecutive inputs at sampling instant p and the preceding instant $p-1$. For example, the event occurs at time p if the input tapes are

1

0	1	$p=2$
ρ_3	ρ_5	ρ_3

2

0	1	2	3	4	$p=5$
ρ_2	ρ_1	ρ_7	ρ_4	ρ_5	ρ_3

3

0	1	2	3	4	5	6	7	8	$p=9$
ρ_4	ρ_1	ρ_5	ρ_3	ρ_7	ρ_5	ρ_3	ρ_2	ρ_5	ρ_3

4

0	1	2	3	4	5	6	7	8	9	$p=10$
ρ_3	ρ_5	ρ_3	ρ_5	ρ_3	ρ_5	ρ_3	ρ_5	ρ_3	ρ_5	ρ_3

and it does not occur with, for example, the input tapes

5

0	1	$p=2$
ρ_5	ρ_3	ρ_5

6

0	1	2	3	4	$p=5$
ρ_2	ρ_1	ρ_7	ρ_4	ρ_3	ρ_5

7

0	1	2	3	4	5	6	7	8	$p=9$
ρ_4	ρ_1	ρ_5	ρ_3	ρ_7	ρ_5	ρ_3	ρ_2	ρ_4	ρ_1

8

0	1	2	3	4	5	6	7	8	9	$p=10$
ρ_3	ρ_5	ρ_3	ρ_5	ρ_3	ρ_5	ρ_3	ρ_5	ρ_3	ρ_3	ρ_5

If the tape is

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	...
ρ_3	ρ_1	ρ_4	ρ_8	ρ_6	ρ_5	ρ_3	ρ_7	ρ_9	ρ_1	ρ_5	ρ_3	ρ_5	ρ_3	ρ_5	ρ_2	ρ_4	ρ_3	ρ_5	ρ_5	ρ_3	...

then the event occurs at $p = 6, 11, 13$, and 20, and it does not occur at all other p's.

Example 2. The event occurs if prior to the sampling instant p there is at least one ρ_5, ρ_3 input sequence. This condition is met by all the above tapes except 6; therefore, after some initial time, all these tapes belong to subject G^*. Thus tape 2 belongs to G^* at $p \geqslant 5$, tape 3 at $p \geqslant 3$, tape 4 at $p \geqslant 4$, and so on.

In other typical formulations, the event occurs if:

Example 3. The input tape contains the sequence $\rho_2 \rho_6 \rho_1$ between p-5 and p.

Example 4. At least one of the sequences $\rho_1\rho_2\rho_6$, $\rho_2 \rho_4 \rho_5$, or $\rho_3 \rho_2 \rho_1$ appears on the tape between p-2 and p.

Example 5. There is no input ρ_3 between p-2 and p.

Example 6. No input ρ_1 is encountered between p-4 and p unless it follows ρ_3.

Example 7. Input ρ_5 appears at least once prior to p.

Example 8. Input ρ_7 is followed by ρ_3 at least once prior to p.

Example 9. There is no ρ_7 prior to p.

Example 10. Prior to p, there is no input sequence of three symbols ρ with odd subscripts after a ρ with an even subscript.

Example 11. At $p = 2$, the input is ρ_7.

Example 12. At $p = 4$, the input is ρ_1, ρ_6, or ρ_2.

Example 13. At $p = 2, 6$, and 8 the input is a ρ with an odd subscript.

Example 14. The input value is divisible by three (e.g., at $p = g$).

Example 15. $p = k$.

Note: In this case, the set G^* is equal to the set of all tapes of length k.

Example 16. A symbol ρ with an even subscript is the input at all p's whose value is a square of an integer (e.g., 1, 4, 9, 16, 25, ...).

Example 17. Either ρ_5 or ρ_9 appears at the input whenever the p is a prime number (e.g., 2, 3, 5, 7, 11, and so on).

In considering the occurrence of events we assumed that there is someone examining the input tape and deciding whether the sequence of inputs ρ appearing on the tape at time p belongs to the given set G^* or not. Now, couldn't this very task be performed by an automaton or an s-machine? To formulate this question precisely, we shall introduce the concept of representation of events by an automaton and an s-machine.

Consider the set K of all possible states $\varkappa_1, \varkappa_2, \ldots, \varkappa_k$ of a finite automaton (which at $t = 0$ is in an initial state $\varkappa^0$), and select a nonempty subset M (in particular, M may contain only one state). We shall say that *this automaton represents a given event G^* by states from set M if, and only if, at $t = p + 1$ it is in a state of set M solely because of occurrence of the input event G^* at $t = p$.*

If the automaton is associated with an output converter, the latter can always be so designed that it generates an output of 1 when $\varkappa$ belongs to M, and a 0 for other $\varkappa$. In this case the automaton represents an event if, and only if, it generates an output or 1 at $t = p + 1$ solely as a result of an event occurring at $t = p$.

Similarly, a sequential machine represents an event by generating an output symbol 1 at $t = p + 1$ if, and only if, the event occurs at the input at $t = p$.

We shall say that an event G^* is representable in a finite automaton if there exists a finite automaton A that represents the event G^*.

There is an important consequence of the general theorem proven in Section 4.3: from the representability (or nonrepresentability) of an event by an automaton follows its representability (or nonrepresentability) by a sequential machine. The converse statement is also true. For this reason, in discussing representability of events, we need only to consider the case of finite automata.

7.3. OPERATIONS ON SETS OF INPUT SEQUENCES

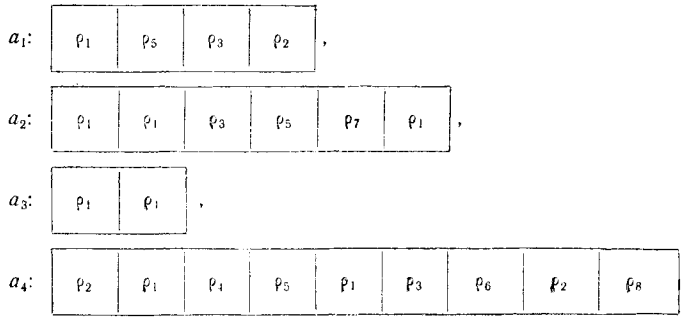
Regular Events

So far, when speaking of an input tape we meant the top strip of the tape of the automaton, that is, the sequence of input symbols

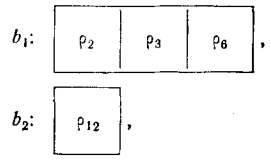
associated with a discrete time sequence and starting from $t = 0$. However, in the first part of this section we shall deal merely with sequences of input symbols ρ without in any way associating the beginning and the end of any such sequence with a specific time. We shall denote specific, finite input sequences by a, b, c, and so on or a_1, a_2, a_3, and so on.

Let A and B be sets (finite or infinite) of input sequences, composed of elements $a_1, a_2, \ldots$ and $b_1, b_2, \ldots$ respectively. We shall form new sets from sets A and B by means of three operations.

First operation: *union of sets.* The set C, containing all sequences of set A and of set B, will be called the *union* (or sum) of A and B and will be denoted by $C = A \vee B$. Thus, for example, if the set A consists of the four sequences



and the set B of the two sequences



then the set C will consist of all the above six sequences; that is,

$$c_1 = a_1, \quad c_2 = a_2, \quad c_3 = a_3, \quad c_4 = a_4, \quad c_5 = b_1, \quad c_6 = b_2.$$

Second operation: *multiplication.* From the sequences contained in sets A and B, we form a new set of sequences via the following rule: we add to the right-hand side of any sequence of set A, for example, a_i, any sequence of set B, for example, b_j, thus forming a new sequence $c_k = a_i b_j$. Let us form all possible sequences of this type, in turn adding to each sequence of A one of the possible sequences

of B. The new set of sequences C is

$$c_k = a_i b_j.$$

It shall be called the ***product*** of the sets A and B, and the operation yielding C is the ***multiplication*** of A by B. It is written as

$$C = A \cdot B.$$

If, for example, set A consists of the four and set B of the two sequences of the preceding example, then set $C = A \cdot B$ consists of the following eight sequences:

$c_1 = a_1 b_1 =$ | p_1 | p_5 | p_3 | p_2 | p_2 | p_3 | p_6 |,

$c_2 = a_1 b_2 =$ | p_1 | p_5 | p_3 | p_2 | p_{12} |,

$c_3 = a_2 b_1 =$ | p_1 | p_1 | p_3 | p_5 | p_7 | p_1 | p_2 | p_3 | p_6 |,

$c_4 = a_2 b_2 =$ | p_1 | p_1 | p_3 | p_5 | p_7 | p_1 | p_{12} |,

$c_5 = a_3 b_1 =$ | p_1 | p_1 | p_2 | p_3 | p_6 |,

$c_6 = a_3 b_2 =$ | p_1 | p_1 | p_{12} |,

$c_7 = a_4 b_1 =$ | p_2 | p_1 | p_4 | p_5 | p_1 | p_3 | p_6 | p_2 | p_8 | p_2 | p_3 | p_6 |,

$c_8 = a_4 b_2 =$ | p_2 | p_1 | p_4 | p_5 | p_1 | p_3 | p_6 | p_2 | p_8 | p_{12} |.

Third operation: *iteration.* The first two operations were binary; that is, a new set C was formed from two sets A and B. The third operation is unary; that is, it forms a new set C from some single set.

Consider, for example, set B, and select a sequence from it. Then add to its right side any sequence from the same set B (it may also be the first sequence itself). Then add to this new sequence another sequence from set B, and so on, any (finite) number of times.

This process of adding sequences from set B one after the other may be interrupted at any point, in particular after the first step, that is, after selection of the first sequence from B. At any time during this process one can select any sequence from B, including any of those already utilized before.

In forming all possible new sequences from sequences belonging to B, that is, in running through all the possibilities of attaching one sequence of B to another sequence from B, and interrupting this process at every conceivable finite step, we form a new set of sequences C. If we now add to set C an "empty"* sequence Λ (containing no symbols), we shall obtain set C', which is known as the *iteration* of B, and is written as

$$C' = B^*.$$

Even if set B is finite, set $C' = B^*$ is infinite. Thus in our example set B consisted of two elements

b_1: | p_2 | p_3 | p_6 |,

b_2: | p_{12} |.

The elements of the infinite set $C' = B^*$ are, for example, the sequences

$c_0 = \Lambda$ ("empty" sequence),

$c_1 = b_1$: | p_2 | p_3 | p_6 |,

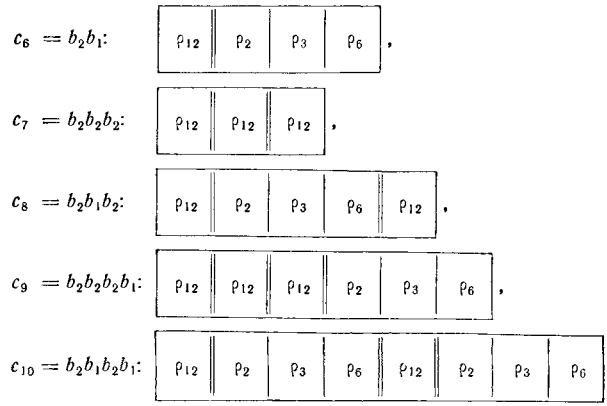
$c_2 = b_2$: | p_{12} |,

$c_3 = b_1b_1$: | p_2 | p_3 | p_6 | p_2 | p_3 | p_6 |,

$c_4 = b_1b_1b_1$: | p_2 | p_3 | p_6 | p_2 | p_3 | p_6 | p_2 | p_3 | p_6 |,

$c_5 = b_1b_2$: | p_2 | p_3 | p_6 | p_{12} |,

*Multiplication of any sequence a by an "empty" sequence Λ yields a: $a\Lambda = \Lambda a = a$. Introduction of "empty" sequences is convenient for writing of regular expressions (see below, Example 2).



and so on.

We now see the relationship between multiplication and iteration: iteration is the result of union of all the sets obtained by multiplying set B by itself some (finite) number of times. Accordingly, iteration may be represented as an "infinite series"*

$$B^* = \Lambda \vee B \vee (B \cdot B) \vee ((B \cdot B) \cdot B) \vee (((B \cdot B) \cdot B) \cdot B) \vee \ldots$$

The new sets generated from A and B can also be treated as initial sets: they can be operated upon to form new sets, and so on. Even if the initial sets are finite (and even if each contains only one sequence consisting of one symbol), one iteration operation will produce an infinite new set.

Let us introduce a *universal set* E, containing some input sequence of some (finite) length.

If the set $\bar{A}$ is a set of sequences containing only one symbol ρ, that is, if it is assumed that $\bar{a}_1 = \rho_1$, $\bar{a}_2 = \rho_2$, ..., $\bar{a}_r = \rho_r$, then the universal set E can be obtained from the elements of $\bar{A}$ by adding these one after another in any desired number and in any desired order, that is,

$$E = \bar{A}^*.$$

*These three operations just discussed were first introduced by Kleene [42]. Our exposition differs from that of Kleene in two respects: 1) Instead of the unary operation of iteration, Kleene uses the binary operation $C = A * B$, which may be expressed by means of our three operations: $A * B = A \vee A \cdot B^*$; 2) In contrast to Kleene, we add sequences during multiplication and iteration on the right- and not the left-hand side.

In the following we shall call *basic** a) any sets consisting of one input sequence of finite length (a, b, c, ...) and b) the universal set E.

We shall call a *regular set* of input sequences:

1) any basic set;

2) any set of sequences that may be formed from the basic ones by using union of sets, multiplication, or iteration over a finite number of times.

Regular sets, of which a few examples follow, will be denoted by R.

Example 1.

$$R = [(a \vee b) \vee c].$$

In this case the regular set is the union of the three basic sequences.

Example 2.

$$R = [b \cdot (a)^*].$$

The regular set is the set of all sequences starting with b, followed only by element (sequence) a, which may be repeated any number of times. For example:

$$b,\ ba,\ baa,\ baaa, \ldots\ .$$

Note that because the definition of $(a)^*$ includes empty sequence Λ, set R also contains symbol b by itself.

Example 3.

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

This set contains all sequences consisting of sequences a and b repeated any desired number of times and in any order, and terminating in sequence c. For example:

$$c,\ abc,\ bac,\ aabc,\ baabc,\ baaaabbc\ ,$$

and so on.

*It should be pointed out that it would be logical to treat as basic sequences not those of finite length but sequences of length 1: $\bar{a}_1 = \rho_1$, $\bar{a}_2 = \rho_2$, ..., containing only one symbol. Indeed, we can obtain finite sequences from these merely by multiplication. However, in proving Kleene's theorems, we shall find that basic sequences of finite length are more convenient. Later, in Chapter 8, we shall give another proof of Kleene's theorems, and there we shall use basic sequences containing only one symbol.

Example 4.

$$R = (E \cdot a).$$

Here R is the set of all sequences terminating in sequence a.

Example 5.

$$R = |(E \cdot a) \cdot b|.$$

This set contains all sequences terminating in a sequence of sequences a and b.

Example 6.

$$R = \{|(a \cdot E) \cdot (c \cdot E)| b\}.$$

In this case R contains all sequences starting with a, terminating in sequence b, and containing sequence c somewhere in between.

Expressions such as in the examples, that is, formed from the basic sequences (a, b, c, ... and E) connected by the signs for union of sets, multiplication, and iteration ($\vee$, $\cdot$, and *), shall be called *regular expressions*.

In regular expressions, each sign for an operation may be used only with a pair (in binary operations) or a single basic element (in iteration), or with parentheses (brackets) containing the result of such an operation. For this reason, a regular expression may contain "parentheses within brackets" (see examples above).

The highest number of "parentheses (brackets) within brackets" in a regular expression (counting the external brackets) is known as the depth of the regular expression.* In the above examples the depth is equal to 1 only in Example 4; it is 2 in Examples 1, 2, and 5, and 3 in Examples 3 and 6.

We shall say that the depth of a regular expression is zero if it contains no operations, for example, $R = a$, $R = b$, $R = E$, and so on.

Now, the same regular set of input sequences may be represented by several different regular expressions. For example, expressions

$$R = [(a \cdot b) \cdot (c \cdot d)] \text{ and } R = \{[(a \cdot b) \cdot c] \cdot d\},$$

certainly describe the same set, but they are of different depth. For

*The operations $\vee$, $\cdot$, and * could be used as the basis for algebraic operations on sets of input sequences. In particular, we could obtain identities, for example: $(E \cdot B)^* = EB$ (where B is an arbitrarily given set), which would enable us to simplify regular expressions. However, we shall not need such an algebra in this book.

this reason depth relates to a specific regular expression rather than to a regular set.

Let us also point out that a subset of a regular set of input sequences may not necessarily be regular. This follows from the simple fact that a universal set is regular by definition, whereas irregular sets do exist (for an example of an irregular set, see Section 7.6).

The reader will recall that the sequences treated so far in this section did not carry a number identifying the discrete time at which they appeared at the input.

Let us now consider a set S of such sequences and identify each element of this set with a number describing t. We start with $t = 0$. Then we obtain a set of input tapes from a set of input sequences. Thus, for example, a set consisting of the three input sequences

p_1	p_5	p_{12}	p_3	p_1	p_4	p_6

,

p_2	p_5	p_3

,

p_2	p_1	p_2	p_5	p_6	p_1	p_2	p_8	p_1

,

becomes a set consisting of the three input tapes

Discrete moment	0	1	2	3	4	5	6
p	p_1	p_5	p_{12}	p_3	p_1	p_4	p_6

Discrete moment	0	1	2
p	p_2	p_5	p_3

Discrete moment	0	1	2	3	4	5	6	7	8
p	p_2	p_1	p_2	p_5	p_6	p_1	p_2	p_8	p_1

.

A set of input tapes formed by such a method from a regular set of input sequences will be called a *regular set of input tapes*. We shall associate with each regular set of input tapes the same regular

expression that describes the corresponding regular set of input sequences.

We can now proceed to the classification of events at the input of the automaton.

An input event of the automaton is said to be regular if the set of input tapes (each examined completely from $t = 0$ *to* $t = p$*) defining the occurence of the event is a regular set.*

Among regular events we may distinguish a subclass of events that are described by a regular expression

$$R = E \cdot A,$$

where A is any set of finite input sequences containing not more than q symbols. Such a regular event is known as a specific event *of length* q or simply a *specific event*. With a specific event, one need not examine the entire input tape from $t = 0$ to $t = p$ in order to specify it: one merely looks over a length q of the "tail end" of the tape, corresponding to $t = p, p-1, p-2, \ldots, p-q$. One can visualize this process as one in which the input tape lies under a transparent runner (similar to hairline-carrying runner of a slide rule) through which one can observe only some number q of tape positions. With each discrete instant the runner is displaced one position to the right, so that the extreme right position seen through the runner always corresponds to $t = p$.

Specific events are distinguished by the fact that we can so select the runner (a specific one for each specific event) that at any time p the occurrence or nonoccurrence of an event is indicated simply by those positions on the input tape that can be seen through that runner.

7.4. REPRESENTABILITY OF REGULAR EVENTS

We can now formulate and prove the following fundamental theorems.

Kleene's first theorem. Assuming a suitable initial state of the automaton, any regular event can be represented in a finite automaton equipped with an output converter by generation of 1 at the output of the converter.

We shall prove this theorem by showing one of the possible methods for synthesizing an automaton representing a regular event defined by an arbitrarily chosen regular expression.*

*Another proof of this theorem is given in Chapter 8 in connection with the description of Glushkov's method.

We shall now introduce *auxiliary automata* with output converters (the converter output may only be 0 or 1) having, in addition to input p, auxiliary inputs for symbols α from the alphabet {0, 1}.*

Assume that a regular set of input sequences R is given. We shall say that an *event* R_α occurs at the input of the auxiliary automaton at time p if there exists a time t $(0 \leqslant t \leqslant p)$ such that:

1) the symbol $\alpha = 1$ appears at the auxiliary input at t; and

2) the sequence of symbols ρ that appear between times t and p belongs to R.

For example, let the given set R include the three sequences

$$\rho_1\rho_2\rho_3,\ \rho_3\rho_2\rho \text{ and } \rho_5.$$

Then, given the input tape of our auxiliary automaton (Table 7.4), the event R_α will only occur at times 3, 5, 7, 14, and 18.

We shall say that the generation of symbol 1 at the output of the converter for the auxiliary automaton represents the event R_α if, and only if, the event R_α occurs at the input of this automaton.

Table 7.4

Discrete moment	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
ρ	ρ_1	ρ_3	ρ_2	ρ_1	ρ_2	ρ_3	ρ_5	ρ_5	ρ_3	ρ_4	ρ_5	ρ_2	ρ_1	ρ_2	ρ_3	ρ_5	ρ_3	ρ_2	ρ_1
Auxiliary input	0	1	1	1	0	1	0	1	1	0	0	0	1	1	1	0	1	0	1

Now let us imagine that there is an autonomous automaton with an output alphabet {0, 1} and that the output of this automaton coincides with the wire α of the auxiliary automaton (Fig. 7.1). The whole network so obtained constitutes an automaton with a single input ρ. Further, let the autonomous automaton generate an output of $\alpha = 1$ at time zero, and $\alpha = 0$ at all subsequent times. Then our network will represent the event R by generating an output of 1.

Such an autonomous automaton can, indeed, be synthesized: this may be, for example, a binary delay element whose input is always zero, while its initial state is 1. Therefore, if we can show that an auxiliary automaton representing the event R_α can be synthesized

*In other words, the input symbols of such an automaton are symbol pairs $(\rho, 1)$ or $(\rho, 0)$.

regardless of the type of regular event R, then we shall have proven the existence of an automaton representing any regular event R. Our proof will be inductive and be based on the depth of the regular expression defining the regular event R_α being represented. We shall show first a way of representing all regular events R_α with a zero depth of the defining regular expression R. Then we shall prove that if a regular event R_α defined by a regular expression R of arbitrary depth ν can be represented, then any regular event R_α defined by a regular expression R of depth $\nu + 1$ can also be represented.

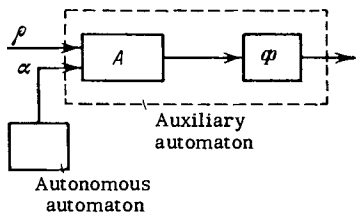


Fig. 7.1.

First inductive step. Recall that regular expressions of depth zero are simply symbols corresponding to the given input sequences a, b, c, ..., as well as the symbol E, corresponding to the universal set of input sequences.

Representation of the event R_α denoted by symbol E means devising an automaton with an output of 1 maintained from the first instance when $\alpha = 1$ regardless of what the input sequence may be. Such an automaton may be synthesized from, among others, a binary delay unit that is in its zero state at $t = 0$. The input to this unit, via a disjunction element, is a symbol α and an output symbol* (Fig. 7.2.).

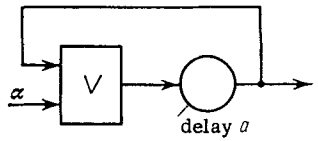


Fig. 7.2.

Let us now construct an automaton representing the event R_α when the set R contains only one input sequence of finite length q. This machine consists of two delay lines (each with q delay units) and a symbol converter (Fig. 7.3). The first (main) delay line operates in alphabet $\{\rho\}$, and the second (auxiliary) in binary alphabet $\{\alpha\}$. The converter output is 1 if, and only if, the symbols ρ at the outputs of all the delay units of the main line form one given sequence of length q, counting from the end of the delay line toward its beginning. The initial state of the delay units in the main line is immaterial, but all such units in the auxiliary line must be in state zero. The output of the automaton is a conjunction of the outputs of the converter and the auxiliary delay line.

The output will be 1 if, and only if: 1) at q discrete instants prior to the sampling instant p there is an input of 1 at α (and therefore

*To make this automaton conform to our other automata, we can assume that it also has an input ρ, but that its output does not depend on ρ.

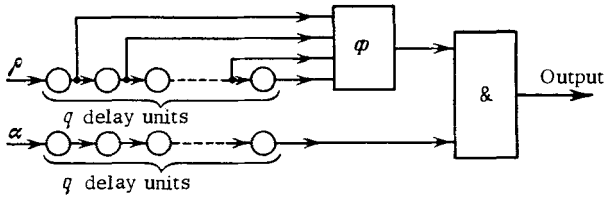


Fig. 7.3.

the auxiliary line has an output of 1 at $t = p$) and 2) there is an input of the given sequence of symbols during the q discrete instants preceding ρ (and therefore there is a converter output of 1 at $t = p$). It is seen from Fig. 7.3 that there cannot be an output of 1 until q discrete instants after $t = 0$ (since at $t = 0$ all delay units in the auxiliary line are in state zero). It follows from this that the states of the delay units in the main line do not affect the output of the system at $t = p$; that is, q instants after $t = 0$, the states of these delay units are determined only by the inputs.

This completes the first part of our proof. That is, we showed that the system of Fig. 7.3 represents an event R_α where R is any regular expression of depth zero. We now proceed to the second part of the proof.

Induction. We shall now prove that if there exist auxiliary automata that represent events R_α specified by any desired regular expressions of depth ν ($\nu \geqslant 0$), then we can synthesize an automaton that will represent the event R specified by any desired regular expression of depth $\nu + 1$.

From the definition of the concept "depth of a regular expression" it follows that any regular expression of depth $\nu + 1$ is obtained from one or two regular expressions of depth ν via a single multiplication, iteration, or union of sets.

In this connection, and in order to complete the induction, it must be proved that if events R_α, specified by regular expressions R_1 and R_2, are representable, then events R_α, specified by expressions

$$R = (R_1 \vee R_2),$$
$$R = (R_1 \cdot R_2),$$
$$R = (R_1^*) \quad \text{or} \quad R = (R_2^*)$$

are also representable.

Let us consider these three operations separately. For the sake of brevity and wherever there is no risk of ambiguity, we shall denote the regular expression, the event corresponding to it, and the automaton representing the event R_α by the same letter R.

Union of sets. An automaton representing a union

$$R = (R_1 \vee R_2),$$

is obtained from two automata, respectively representing R_1 and R_2. This is done by connecting their inputs and feeding their outputs to a common disjunction element (Fig. 7.4).

Multiplication. An automaton representing a product

$$R = (R_1 \cdot R_2),$$

is obtained from two automata, representing R_1 and R_2, respectively, by feeding the output of automaton R_1 into the auxiliary input of R_2 via a delay unit (Fig. 7.5). Then the auxiliary input line of R_1 becomes the auxiliary input to the system, while the output of R_2 is the output of the system. Indeed, the output of R_2 will be 1 if R_2 represents an event that has begun during the instant following an output of 1 in R_1. But an output of 1 in R_1 means that the event R_1 had already been represented prior to that instant, and that this event had begun at the instant of occurrence of 1 at the auxiliary input of R_1 (and therefore at the input of our entire automaton system). Consequently the automaton system as a whole represents the event R_α which is described by: "the event R_2 occurred directly after the event R_1," that is, the product $R_1 \cdot R_2$.

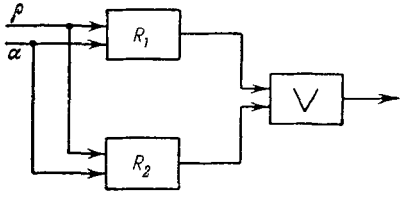


Fig. 7.4.

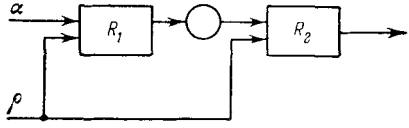


Fig. 7.5.

Iteration. If the automaton R_1 representing the event R_1 is given, then the automaton representing an R_α event: $R = (R_1^*)$ may be obtained by coupling R_1 to one disjunction element (Fig. 7.6). The disjunction element output is made to coincide with the auxiliary input to R_1, while the input of disjunction element consists of the auxiliary input α to R_1 as well as the R_1 output, which is made to pass through a delay unit. Indeed, the output of this system is 1 whenever the event R_1 is represented, starting with the appearance of 1 at the auxiliary input α, or from the instant directly following the representation of the event R_1. Therefore, there is a system output of 1

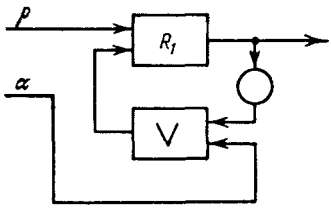


Fig. 7.6.

whenever the event R_1 occurs (this can happen any number of times in succession), which is what indeed constitutes the representation of event $R = (R^*)$.

This concludes the induction process based on the depth of the regular expression, and therefore completes the proof of the entire theorem.

First note. Our formulation of Kleene's first theorem included the statement about the choice of a suitable initial state. It can be seen from the proof that a suitable choice of the initial state reduces to: a) ensuring a state of 0 in all units of the auxiliary delay line during representation of the given set consisting of one input sequence (Fig. 7.3); b) ensuring a state of 1 in the delay unit of the automaton representing a universal event (Fig. 7.2); and c) ensuring a state of 1 in the delay unit of the autonomous automaton representing the event $t = 0$ (see above).

Second note. The representation of the event ER differs from that of the event R only in the method of utilizing the auxiliary input of the automaton which represents the event R_a. To be precise, in order to represent the event R, this input is connected to the autonomous automaton which produces 1 only at $t = 0$; however, in order to represent the event ER, this input consists of the output of the automaton representing the event E (Fig. 7.2); that is, this input is always 1. This applies, in particular, to the representation of a specific event $E \cdot a$.

7.5. REGULARITY OF REPRESENTABLE EVENTS

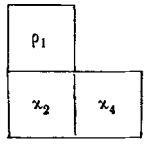
In this section we shall prove a theorem that is the converse of the one proved above.

Kleene's second theorem. Only regular events are representable in a finite automaton.

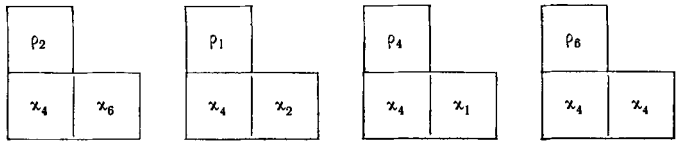
To prove this theorem we shall first introduce the concept of regular sets of triad chains, then we shall prove an auxiliary lemma, and finally with the help of this lemma, we shall prove the theorem.

The triad labyrinth and labyrinthine paths. The infinite set of tapes which may be generated in a finite automaton contains only a finite number kr of different triads. Let us denote these by μ_1, μ_2, ..., μ_{kr} and match each triad μ_i to a point in a plane.

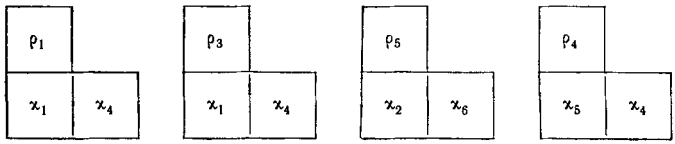
Two adjacent triads must have one common symbol. For this reason, the triad



must be followed by a triad containing x_4 at lower left, for example,



so that triads with any other symbols in this position, for example,



are not allowed.

Now, we draw arrows from the point corresponding to triad μ_i to all points corresponding to triads which can follow μ_i on the tape (Fig. 7.7); we do the same for all triads μ_i ($i = 1, 2, \ldots, kr$). We obtain (see Fig. 7.8) a plane diagram, which we shall call the *triad labyrinth*.

Now let some triad μ_i be the starting point from which we proceed along any desired arrow, writing out the sequence of triads encountered on the way. This sequence is called the *path in the triad labyrinth*, or simply a *path*. The length of a path is measured by the number of triads contained in it. Thus, if μ_2 is the starting point, Fig. 7.8 has paths described by triad chains.

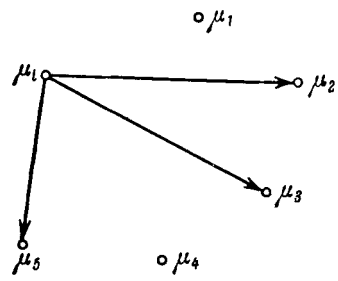


Fig. 7.7.

$\mu_2\ \mu_3\ \mu_4\ \mu_5\ \mu_6\ \mu_1\ \mu_2$,
$\mu_2\ \mu_4\ \mu_3\ \mu_1\ \mu_4\ \mu_5$,
$\mu_2\ \mu_1\ \mu_3\ \mu_4$, etc.

However, path

$$\mu_2\ \mu_3\ \mu_6$$

is not allowed since there is no arrow from μ_3 to μ_6.

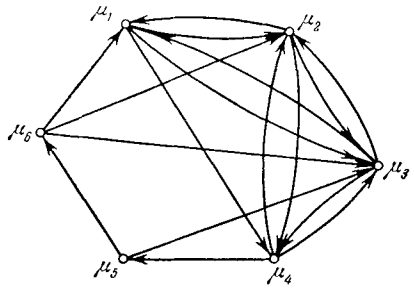


Fig. 7.8.

Now let us proceed to the concept of a *regular set of triad chains*. By definition, all sets containing only one triad are regular; if A and B are regular sets of triad chains, then, by induction, the following are also regular:

a) their union $(A \vee B)$, that is, a set containing both A and B;

b) their product $(A \cdot B)$, that is, a set of chains obtained by adding some chain from B to the right of each chain from A;

c) their iteration (A^*), that is, a set of chains obtained by adding to the right-hand sides of all chains from A any and all chains from A (including itself or any of those already added) and doing that any desired number of times.

Lemma. The set of all paths leading from triad $\bar{\mu}$ to triad $\tilde{\mu}$ in the labyrinth of any finite automaton is a regular set of triad chains.

We shall prove this lemma by induction based on q, where q is the number of different triads on the path from $\bar{\mu}$ to $\tilde{\mu}$.

For $q = 1$, the lemma is obvious: μ coincides with μ and only two subcases are possible:

a) if all symbols $\varkappa$ in the lower line of this triad $\bar{\mu} = \tilde{\mu}$ are not the same, then the set of paths consists of only one path containing this one triad $\bar{\mu}$;

b) if all $\varkappa$ in $\bar{\mu}$ are the same, then the set of paths comprises all triad chains repeating the same triad $\bar{\mu}$, that is, the iteration $(\bar{\mu})^*$.

Consequently the set is regular for $q = 1$.

Now let the lemma hold for any set of paths with $q \leqslant m$. We shall prove that it also holds for any set of paths with $q = m + 1$. Thus let us consider any path where $q = m + 1$. We write out all the triads $\bar{\mu}$ as well as the triad $\tilde{\mu}$ at the end of the path, and replace the remaining triads of this path (including any triads $\tilde{\mu}$ that may occur at points other than the end) by groups of dots:

$$\bar{\mu} \ldots \bar{\mu} \ldots \bar{\mu} \ldots \bar{\mu} \ldots \bar{\mu} \ldots \tilde{\mu}.$$

Each group of dots stands for a path that does not contain the triad $\bar{\mu}$. For this reason, the number of different triads contained in each path replaced by a group of dots is smaller than $m+1$ by at least 1, that is, $q \leqslant m$. By induction, each path replaced by dots is a regular set of triad chains. We shall denote these replaced paths by $\bar{r}_i$, $\bar{r}_j$, $\bar{r}_g$, and so on.

To start with, let $\tilde{\mu} = \bar{\mu}$. Then one can imagine the entire path from $\bar{\mu}$ to $\tilde{\mu} = \bar{\mu}$ as consisting of triads $\bar{\mu}$, with the regular sets $\bar{r}$ interspersed among them. We write this as follows:

$$\bar{\mu}\bar{r}_i\ \bar{\mu}\bar{r}_j\ \bar{\mu}\bar{r}_g\ \bar{\mu}\ \bar{\mu}\bar{r}_s \ldots \bar{\mu}\bar{r}_q\ \bar{\mu}.$$

In this notation, the entire path consists of pairs of consecutive paths. Each such pair is the product of two regular sets: a set consisting of a single element $\bar{\mu}$, and the corresponding set $\bar{r}$. Consequently, such a pair must itself belong to a regular set $\bar{\mu} \cdot \bar{R}$. Therefore, all paths leading from $\bar{\mu}$ to $\tilde{\mu} = \bar{\mu}$ are elements of the iteration of $\mu \cdot \bar{R}$

$$(\bar{\mu} \cdot \bar{R})^* \cdot \bar{\mu};$$

that is, they constitute a regular set of triad chains.

If $\bar{\mu} \neq \tilde{\mu}$ then by the same reasoning the set of all paths leading from $\bar{\mu}$ to $\tilde{\mu}$ is

$$(\bar{\mu} \cdot \bar{R})^* \cdot \tilde{\mu};$$

that is, it is also regular. The lemma is thus proved.

This lemma can be readily explained on a triad labyrinth.

Consider the triad labyrinth of Fig. 7.9 (where some arrows have been omitted for clarity), and run through the possible paths from $\bar{\mu}$ to $\tilde{\mu}$.

First, there is a choice of the two paths

$$\bar{\mu}\ 1\ 2\ \ 3\ \tilde{\mu}$$

and

$$\bar{\mu}\ 8\ 9\ 10\ \tilde{\mu}.$$

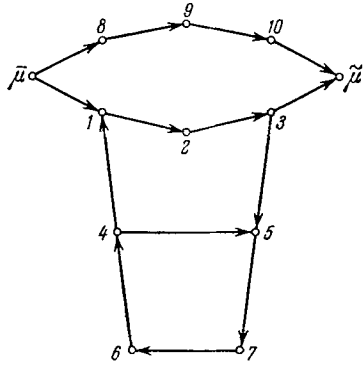


Fig. 7.9.

Second, the first path may be lengthened by including a loop, for example,

$$\bar{\mu}\ 1\ 2\ 3\ 5\ 7\ 6\ 4\ 1\ 2\ 3\ \tilde{\mu}.$$

The looping paths may be traversed any desired finite number of times, for example,

$$\bar{\mu}\ 1\ 2\ 3\ 5\ 7\ 6\ 4\ 1\ 2\ 3\ 5\ 7\ 6\ 4\ 1\ 2\ 3\ 5\ 7\ 6\ 4\ 1\ 2\ 3\ \tilde{\mu},$$

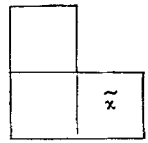
or the path may be complicated still further by including "loops within loops," for example,

$$\bar{\mu}\ 1\ 2\ 3\ 5\ 7\ 6\ 4\ 1\ 2\ 3\ 5\ 7\ 6\ 4\ 5\ 7\ 6\ 4\ 5\ 7\ 6\ 4\ 1\ 2\ 3\ \tilde{\mu},\ \text{etc.}$$

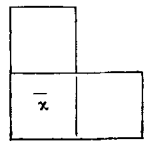
This type of routing is not confined to our example. Each set of paths from $\bar{\mu}$ to $\tilde{\mu}$ contains, first of all, several variants of simple paths (a union!), and it may also consist of arrows traversed one after another (a product!) and of loops traversed any desired finite number of times along anywhere in the labyrinth (an iteration!). And each path from $\bar{\mu}$ to $\tilde{\mu}$ may consist of these three components. This fact, almost obvious from the examination of the labyrinth, is really what the lemma is stating.

From the lemma immediately follows the proof of Kleene's second theorem.

Proof of Kleene's second theorem. Consider a finite automaton, and write out all its triads (a finite number). Let M be a set of several symbols $\varkappa$ (we shall denote an element of this set by $\tilde{\varkappa}$), and let the automaton (which starts with an initial state $\varkappa^0 = \bar{\varkappa}$) represent an input event by the appearance of a state $\tilde{\varkappa}$. Now, let us select a triad $\tilde{\mu}$ such that the $\tilde{\varkappa}$ in its lower right belongs to M (the other two symbols of the triad need not be part of M):



Let us also select a triad $\bar{\mu}$ containing $\bar{\varkappa}$ in its lower left, the other two symbols being completely arbitrary:



Then, by virtue of the lemma just proved, the set of all paths leading from $\bar{\mu}$ to $\tilde{\mu}$ is a regular set of triad chains.

Consider sets $\bar{A}$ and $\tilde{A}$ of all possible triads $\bar{\mu}$ and $\tilde{\mu}$. The set of all paths connecting any triad $\bar{\mu}$ of $\bar{A}$ with any triad $\tilde{\mu}$ of $\tilde{A}$ is the union of the sets of paths leading from $\bar{\mu}$ to $\tilde{\mu}$; that is, it is regular.

To each triad there corresponds a symbol ρ in the top strip of the tape, and to a chain of triads there corresponds a sequence of ρ. If a set of triad chains is regular then the set of corresponding sequences of ρ is a regular set of sequences, as defined in Section 7.3. Indeed, in devising new paths (from the given ones) by union, multiplication and iteration of chains of triads, we used operations analogous to those introduced in Section 7.3 for obtaining regular sets of sequences, and applied those to the ρ sequences in these chains.

We have not imposed any restrictions on the automaton, that is, on the number and the nature of its triads, or on the choice of the initial state. We have therefore proved that in any automaton, starting from any initial state, the set of input sequences leading from any initial state $\varkappa^0$ to a state comprised in M is regular; that is, we have proved Kleene's second theorem.

7.6. DO IRREGULAR (UNREPRESENTABLE) EVENTS EXIST?

Irregular events, which cannot be represented in an automaton, do exist, as we shall prove by examples. Moreover, we shall show that there exists an entire class of events, *a priori* known to be irregular.

Suppose we are given an infinite sequence of ρ, for example,

$$\rho_5\ \rho_2\ \rho_1\ \rho_5\ \rho_4\ \rho_3\ \rho_2\ \rho_1\ \cdots$$

which we shall call $\mathscr{L}$. We will say that $\mathscr{L}$ is *ultimately periodic* if, starting with the qth symbol from the left, the sequence contains a periodically recurring segment of finite length.

Consider an event such that the input sequence of ρ coincides at at all times p with the segment of $\mathscr{L}$ bracketed by its 0th and the pth symbols. If such an event can be represented by an automaton, we shall say that *the automaton represents sequence* $\mathscr{L}$.

Theorem. A given infinite sequence $\mathscr{L}$ *of input symbols* p *is representable in a finite automaton if and only if* $\mathscr{L}$ *is "an ultimately periodic sequence."*

Proof. First, we shall prove that all ultimately periodic sequences are representable. Let such a sequence consist of an "initial segment" A of finite length h and then a periodically recurring "segment" B of finite length T. Consider all the initial sections

$B_1, B_2, \ldots, B_{T-1}$ of segment B (B_1 contains only the first symbol of B; B_2 contains the first two symbols, and so on), as well as the initial sections $A_1, A_2, \ldots$ of segment A. Thus the ultimately periodic sequence—that is, the input event—really consists of segment A followed by segment B, which is then repeated a number of times; the sequence terminates in one of the sections $B_1, B_2, \ldots, B_{T-1}, B$. Therefore $\mathscr{S}$ can be written as a regular expression

$$R = \{[A(B)^*] \cdot (B_1 \vee B_2 \vee \ldots \vee B_{T-1} \vee B)\} \vee (A_1 \vee A_2 \vee \ldots),$$

and, by virtue of Kleene's first theorem, such an event is representable in a finite automaton.

Now let us prove the converse statement, that is, that all representable sequences are ultimately periodic. Let sequence $\mathscr{S}$ be represented by a finite automaton which has k internal states. With $\mathscr{S}$ as the input, let us examine what happens during the first $k + 1$ sampling instants. Since $t = k + 1$, and the machine can have only k states, there must be at least one internal state $\bar{x}$ that will be repeated at least twice during this time. Let that occur in instants t' and t'' ($0 \leqslant t' < t'' \leqslant k + 1$). Now let us compile a sequence $\overline{\mathscr{S}}$, which differs from $\mathscr{S}$ in that the segment from t' to t'' has been thrown out; we then make $\overline{\mathscr{S}}$ the input of the same automaton. Thus, if with $k = 8$ sequence $\mathscr{S}$ corresponds to a tape in which $t' = 3$, $t'' = 6$ (Table 7.5), the tape for $\mathscr{S}$ then would be that of Table 7.6.

Table 7.5

t	0	1	2	$t' = 3$	4	5	$t'' = 6$	7	8	9	...
p	p_1	p_3	p_5	p_1	p_7	p_6	p_{12}	p_{10}	p_1	p_5	...
x	...	...	...	$\bar{x}$	...	...	$\bar{x}$	...	...	...	...

Table 7.6

t	0	1	2	3	4	5	6	7	8	9	...
p	p_1	p_3	p_5	p_{12}	p_{10}	p_1	p_5	...	...	...	...
x	...	...	...	$\bar{x}$	...	...	...	...	...	...	...

After t'' (that is, from instant 7), the sequence of $\varkappa$ on the first tape coincides with the $\varkappa$ sequence appearing on the second tape after t' (that is, from instant 4). This is because from these instants on, the inputs ρ are the same in both cases, as are the initial states $\varkappa$ of the automaton. And since $\mathscr{S}$ is represented by the automaton, $\overline{\mathscr{S}}$ must also be represented by it. This, however, does not contradict our definition of representability, according to which $\mathscr{S}$ and $\overline{\mathscr{S}}$ can be represented by the same automaton only if they coincide. For two infinite sequences of symbols, $\mathscr{S}$ and $\overline{\mathscr{S}}$, where $\overline{\mathscr{S}}$ differs from $\mathscr{S}$ by the fact that it lacks all symbols between t' and t'', may coincide only if beginning from t', $\mathscr{S}$ periodically repeats the segment occurring between t' and t''.

The theorem is thus proved.

This theorem enables us to specify an infinite number of examples of unrepresentable events, but we shall give only two such examples.

Example 1. An event occurs if the automaton input at $t = \alpha^2$ ($\alpha = 1, 2, 3, \ldots$) is ρ_1, and if the input is ρ_2 at all other times.

Example 2. An event occurs if the number of symbols ρ_2 between two nonadjacent symbols ρ_1 of the input sequence always doubles. Thus the event takes place if the sequence has the form

$$\rho_1\,\rho_2\,\rho_1\,\rho_2\,\rho_2\,\rho_1\,\rho_2\,\rho_2\,\rho_2\,\rho_2\,\rho_1\,\rho_1\,\rho_2\,\rho_2\,\rho_2\,\rho_2\,\rho_2\,\rho_2\,\rho_2\,\rho_2\,\rho_1\,\rho_1\,\rho_1\,\ldots,$$

and does not take place when the sequence is

$$\rho_1\,\rho_2\,\rho_1\,\rho_2\,\rho_2\,\rho_1\,\rho_2\,\rho_2\,\rho_2\,\rho_2\,\rho_1\,\rho_1\,\rho_2\,\rho_2\,\rho_1\,\ldots\,.$$

The events of each of the above examples are unrepresentable since the corresponding sequences are not ultimately periodic.

It must not be assumed, however, that all not representable events are embraced by this theorem. For example, we may have an input alphabet ρ consisting of two symbols $\{0, 1\}$, and we may define an event so that at $t = p$, the number of symbols 1 in the output sequence is equal to the number of symbols 0. This event does not belong to the class considered above (since the occurrence of this event is determined not by one specific sequence but by an entire class of such sequences), but at the same time it is not representable. We shall prove this.

Assume we are given some finite automaton A, and that its input sequence consists of zeros only. Then at least one internal state $\varkappa$ of the automaton must sooner or later recur. Let that happen at t' and t'' ($0 \leqslant t' < t''$). Now consider the following two input sequences:

1) $\mathscr{L}$, consisting of t'' zeros followed by t'' symbols 1 and 2) $\mathscr{L}^*$, consisting of t' zeros followed by t'' symbols 1. The corresponding tapes are shown in Tables 7.7 and 7.8.

Table 7.7

t	0	1	2	...	$t'-1$	t'	...	$t''-1$	t''	...	$2t''-1$
ρ	0	0	0	...	0	0	...	0	1	...	1
$\varkappa$	...	...	...	...	...	...	...	...	$\bar{\varkappa}$	...	$\varkappa^*$

Table 7.8

t	0	1	2	...	$t'-1$	t'	...	$t'+t''-1$
ρ	0	0	0	...	0	1	...	1
$\varkappa$	...	...	...	...	...	$\bar{\varkappa}$	...	$\varkappa^*$

Beginning at t' and ending at $t'+t''$, the sequence of $\varkappa$ in the $\mathscr{L}^*$ tape coincides with that $\varkappa$ sequence of the $\mathscr{L}$ tape which starts at t'' and ends at $2t''$. This is because the initial states (at t' and t'', respectively) and the input sequences (t'' symbols 1 in succession) coincide in the two cases. Therefore, given tape $\mathscr{L}^*$, the automaton will attain state $\varkappa^*$ at $t'+t''$; this state will coincide with the state occurring at $2t''$ with tape $\mathscr{L}$. However, the event does occur with input sequence $\mathscr{L}$ (the number of symbols 1 is equal to the number of zeros), whereas it does not occur with $\mathscr{L}^*$ (the number of symbols 1 exceeds that of zeros), even though the automaton A achieves the same state in both cases. This means that automaton A cannot represent this event. Q.E.D.

It is easy to perceive why the events of these examples can not be represented. The reason is that, by definition, the finite automaton (with a finite number of states) has a "finite memory." But in the above examples, the amount of information which the machine must "remember" to be able to "decide" at any given instant whether an event does take place, becomes infinite with time (in the first example, the machine must "remember" how many instants have elapsed prior to sampling; in the second example, the machine

must "count" the zeros occurring between two nonconsecutive symbols of 1, and this number also goes to infinity).

Let us point out that a set of sequences specifying an irregular (nonrepresentable) event may itself be a subset of a regular set of sequences. For example, an event specified as "two symbols 1 never appear consecutively at the input" is representable. However, the input sequences of Example 1 (see above) which are a subset of this regular set, constitute an irregular set.

7.7. WHAT A FINITE AUTOMATON "CAN DO"

In the preceding chapter we found out what an autonomous automaton "can do." Now we have mastered the representability of events and we can thus answer that question for *nonautonomous* automata.

Let A be an automaton with an output converter, or an s-machine representing a regular input event by generating a symbol (say, a 1) at the output. Further, let A stop as soon as 1 is generated at its output and let it trigger an autonomous automaton B that generates a predetermined sequence M. The output of B then becomes the input of automaton C which represents the event M, and the generation of a 1 at the output of C is used to trigger A and stop B (Fig. 7.10). This is done by the automaton Φ, which represents a simple event: the appearance of a given input symbol. Therefore this entire system constitutes a finite automaton. Such a finite automaton responds to any regular event by generating at the output any predetermined finite sequence of states (symbols), after which it is again ready to receive external stimuli, that is, to respond to events.

The automaton of Fig. 7.10, does not respond to those input symbols ρ that appear during the operation of B. If the (discrete) timing of B is so fast that its entire periodic output sequence is generated

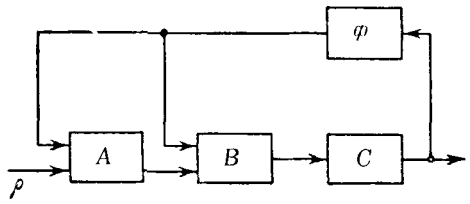


Fig. 7.10.

between two sampling instants of A, then there is no need for automaton Φ, and the output of A can be used as the input of B. Such an automaton maintains an output of any given periodic sequence of symbols throughout the representation of the event, and will generate another predetermined periodic sequence of symbols when no event is represented.

Naturally, other combinations of automata are also possible. But the above two combinations already show that an automaton can respond to any regular event by generating any predetermined cycle.

Can an automaton do something more than this? And if so, what? The answers to such questions depend on what language is used to formulate the laws for the handling of sequences by a finite automaton. Kleene's theorems formulate these laws in the language of representation of events. So far, there are no other convenient languages capable of describing (in terms of necessary and sufficient conditions) what a finite automaton "can do" and what, in principle, it "cannot do." This raises a number of new problems discussed in the next chapter.

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
p	p_0	p_3	p_2	p_3	p_0	p_2	p_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
p	p_3	p_2	p_0	p_0	p_1	p_3	p_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 $(\varkappa_0, \rho_0)$ and $(\varkappa_0, \rho_3)$ 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	$\rho_$	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 $(\varkappa_0, \rho_0)$ 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 $(\rho_2, \varkappa_1)$ 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 $(\rho_2, \varkappa_0)$ 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
$\varkappa$	$\varkappa_0$	$\varkappa_0$	$\varkappa_0$ $\varkappa_3$	$\varkappa_0$ $\varkappa_1$	$\varkappa_0$ $\varkappa_1$	$\varkappa_0$ $\varkappa_1$	$\varkappa_0$ $\varkappa_1$ $\varkappa_2$
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
$\varkappa$	$\varkappa_0$	$\varkappa_1$ $\varkappa_2$	$\varkappa_3$ $\varkappa_0$	$\varkappa_1$ $\varkappa_2$	$\varkappa_1$ $\varkappa_2$
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 $\varkappa$ 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 $(\rho, \varkappa)$ pair which defines a still empty square in the automaton table.

3. If two or more of these last $(\rho, \varkappa)$ 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
$\varkappa$	$\varkappa_0$	$\varkappa_0$	$\varkappa_1$	$\varkappa_0$			
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
$\varkappa$	$\varkappa_0$	$\varkappa_1$	$\varkappa_1$	$\varkappa_0$				
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

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

Table 8.30

Converter Table

$\varkappa$ \ ρ	ρ_1	ρ_2	ρ_3
$\varkappa_0$	λ_1	λ_2	λ_1
$\varkappa_1$	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
$\varkappa_0$	$\varkappa_0$?		$\varkappa_1$	
$\varkappa_1$		$\varkappa_0$		

Table 8.34

Converter Table

$\varkappa$ \ ρ	ρ_1	ρ_2	ρ_3	ρ_4
$\varkappa_0$	λ_1		λ_3	λ_1
$\varkappa_1$		$\varkappa_4$?	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

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

Table 8.38

Converter Table

$\varkappa$ \ ρ	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:

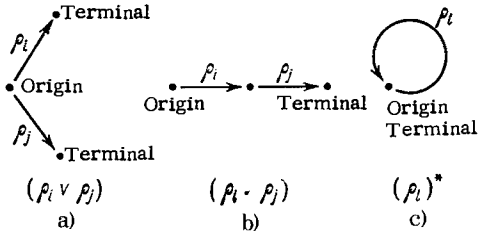


Fig. 8.1.

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).

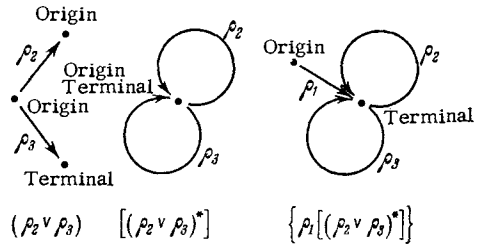


Fig. 8.2.

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.

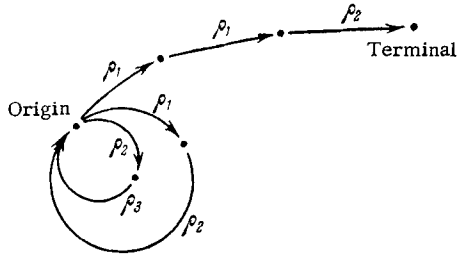


Fig. 8.3.

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.

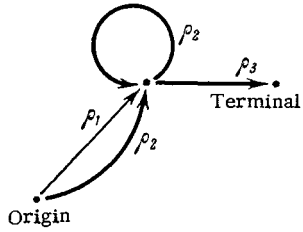


Fig. 8.4.

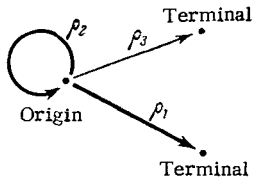


Fig. 8.5.

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

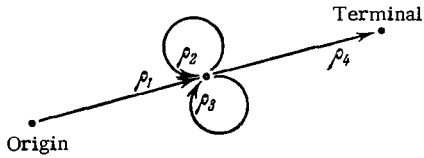


Fig. 8.6.

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

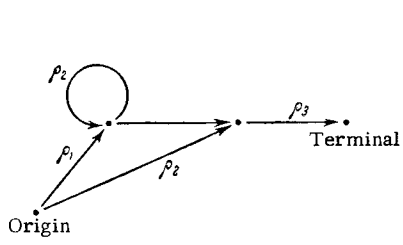


Fig. 8.7.

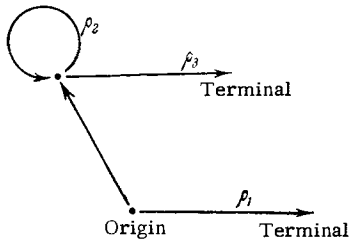


Fig. 8.8.

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.

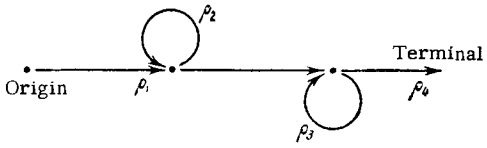


Fig. 8.9.

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

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

Its correct graph is shown in Fig. 8.10.

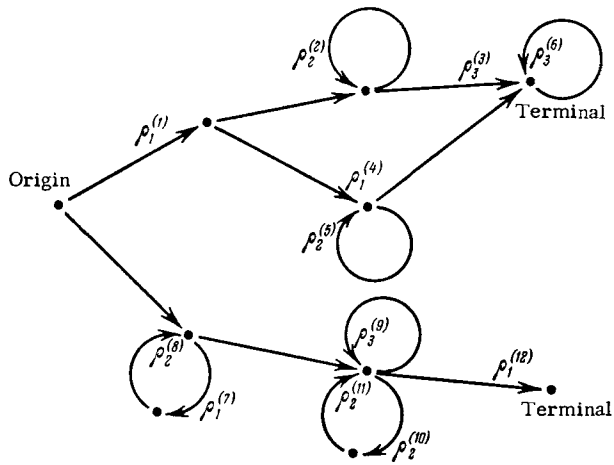


Fig. 8.10.

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

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

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.

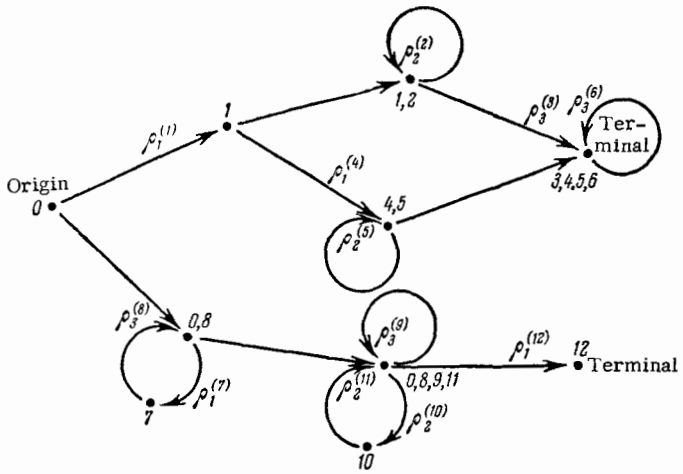


Fig. 8.11.

Table 8.50

Superscript \ ρ	ρ_1	ρ_2	ρ_3
*	*	*	*

Table 8.51

Superscript \ ρ	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_1	ρ_2	ρ_3
*	*	*	*
0	1, 7, 12	10	9
1, 7, 12	4	2	3,8
10	*	11	*
9	12	10	9
4	*	5	6
2	*	2	3
3,8	7, 12	10	6,9
11	12	10	9
12	*	*	*
5	*	5	6
6	*	*	6
3	*	*	6
7, 12	*	*	8
6, 9	12	10	6,9
8	7, 12	10	9

Table 8.55

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

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

$$R_1 = [p_1 \cdot (p_2 \vee p_1)],$$
$$R_2 = [p_2 \cdot (p_3)^*],$$
$$R_3 = \{p_1 \cdot [(p_2)^* \vee p_3]\}.$$

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 ($p_1 \cdot p_2$) 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.

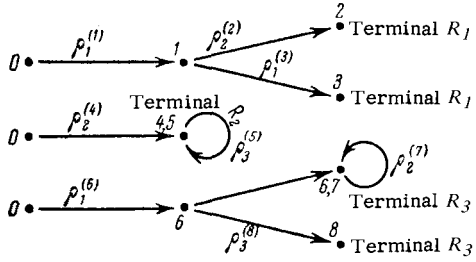


Fig. 8.12.

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

	p \ ρ	ρ1	ρ2	ρ3
	*	*	*	*
	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

	χ \ ρ	ρ1	ρ2	ρ3
	χ_0	χ_0	χ_0	χ_0
	χ_1	χ_2	χ_3	χ_0
λ_3	χ_2	χ_4	χ_5	χ_6
λ_2	χ_3	χ_0	χ_0	χ_7
λ_1	χ_4	χ_0	χ_0	χ_0
λ_1, λ_3	χ_5	χ_0	χ_8	χ_0
λ_3	χ_6	χ_0	χ_0	χ_0
λ_2	χ_7	χ_0	χ_0	χ_7
λ_3	χ_8	χ_0	χ_8	χ_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, ..., 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].

9

Equivalence and Minimization of Sequential Machines

9.1. THE PROBLEM OF RECOGNITION OF EQUIVALENT STATES

We have already said in Section 3.7 that the class of possible input sequences to a machine may be restricted for some reason, and we have seen such a case in Chapter 5. Now consider other constraints that may be imposed on the possible input sequences. They may include the following:

a) Identical symbols shall not appear consecutively.
b) Symbol ρ_j shall not follow symbol ρ_i.
c) An input sequence shall not begin (or, conversely, must begin) with ρ_k.
d) If the sequence contains ρ_s or ρ_t then it cannot contain ρ_q.

In these examples, the infinite set E containing all the possible input symbol sequences of any desired but finite length is split into two subsets: a subset $L \subseteq E$ (which may be finite or infinite), containing all the input sequences allowable in a given s-machine, and a complement of this subset $\bar{L}$, consisting of the set of forbidden input sequences. A special case is that of $L = E$, which means that any input sequence is allowable in the given machine.

Let us note that constraints a - d are in no way related to the state in which the machine happens to be. There may, however, exist other constraints, imposed by the design of the s-machine. For example, its state diagram may show an ith state such as that of Fig. 9.1. Here, the machine cannot accept an input ρ_2, and the constraint on the input sequence is thus imposed by the properties of the machine itself. Such constraints may be imposed on one, several, or even all the states of the machine. The constraints on different states need not

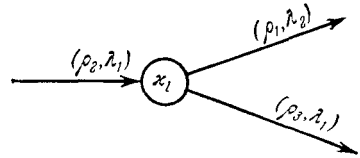


Fig. 9.1.

be identical, and different states may thus forbid different input signals.

If a given input sequence does not violate the constraints imposed by state $\varkappa_i$ and all the states following $\varkappa_i$, it is said to be an *input sequence allowed in state* $\varkappa_i$. The set of all sequences allowed in state $\varkappa_i$ is denoted by $L_{\varkappa_i}$.

Constraints imposed by the properties of the machine are known as *Aufenkamp constraints*. If Aufenkamp constraints are operative, then there is no single, all-encompassing set L such that any member of L would be an allowed input sequence regardless of the initial state of the machine. With Aufenkamp constraints, each of the possible k states of the machine has its own $L_{\varkappa_i}$ which, in general, is different from the other sets $L_{\varkappa_j}$ $(j \neq i)$.

This case differs from that in which the constraints are independent of the states of the s-machine (that is, where there exists a single set L) and where the algorithm for recognizing whether a given sequence belongs to set L can be formulated in terms unrelated to the initial state of the machine (or even completely unrelated to the machine). When the recognition algorithm exists *per se*, that is, may be expressed in terms unrelated to the machine, the corresponding constraints are said to be *constraints per se*, and the set of the possible input sequences is said to be *restricted per se*.

Considering for the time being only sequences restricted *per se*, let us introduce the concept of *equivalence of states* of an s-machine (or a finite automaton). Assume we are given the set L of allowable input sequences, as well as two s-machines S and G (in particular, S may coincide with G). Then state $\varkappa_i$ of S and state $\varkappa_j$ of G are equivalent in terms of L if the two machines (in these two respective states) process the same input sequence from L into identical outputs. If the machines S and G are identical, then this definition merely describes the conditions for equivalence (in terms of L) of the two states of a single s-machine (or automaton). If $L = E$, that is, if the set of allowable input sequences contains all possible sequences, then states $\varkappa_i$ and $\varkappa_j$ are *simply equivalent*.

Our definition of equivalent states underlies the following analytical problem: Given an s-machine and its set of allowable input sequences L, find an algorithm for deciding whether two arbitrarily chosen states of that s-machine are equivalent in terms of L. If we had such an algorithm, we could split the set of all states E into groups of those that are equivalent in terms of L. By a *group of states equivalent in terms of* L we mean a set of states of the s-machine such that: (1) any two states in the group are equivalent in terms of L; and (2) no state from one group is equivalent (in terms

of L) to any other state of any other group. This grouping of states is, as will be seen later, of paramount importance in the minimization of s-machines.

Our generalized analytic problem would be solvable if we had an algorithm for recognizing the equivalence of states, given any s-machine and any set L. But we will show in Section 9.2 that this generalized problem is algorithmically unsolvable, and so we shall be forced to tackle recognition problems one specific case after another, as in Sections 9.3 and 9.4.

9.2. ALGORITHMIC UNSOLVABILITY OF THE GENERALIZED RECOGNITION PROBLEM OF RECOGNITION OF EQUIVALENCE OF STATES

To be useful, the set of allowable input sequences L should be *effectively specified.* In other words, for each specified set L there should exist an algorithm for recognizing whether a given finite sequence of input symbols belongs to L. For example, a finite set L can be effectively specified by simple enumeration of all sequences contained in it. But this cannot be done for an infinite set L, which must be specified in some other way, for instance, by specifying a recognition algorithm. Set L may, for example, be specified verbally by stating that:

1) it contains all sequences longer than three symbols, wherein the fourth symbol is ρ_i; or

2) it contains only those sequences ending in ρ_j which do not comprise ρ_q.

These sets, even though infinite, are fully characterized by their respective verbal descriptions, and thus it is always possible to tell whether they contain any given sequence. The mere fact that such an effective verbal description can be formulated shows that there must exist an algorithm accomplishing the same thing, that is, recognizing whether a given sequence belongs to the given set L. In this sense, the recognition algorithm is the least artificial and the broadest language for effective definition of infinite sets L.

We shall now try to ascertain whether it is possible to determine the equivalence of two states with respect to an arbitrary *effectively specified* set L, that is, a set L defined by a recognition algorithm. To start with, we must formalize the concept of a recognition algorithm. As usual, we turn for help to the theory of algorithms and recursive functions,* which asserts that any set of sequences for

*See Chapter 12, and also Section 8.3.

which one can define "recognition rules" is recursive; conversely, one can define such recognition rules for any recursive set (this assertion is a direct result of Church's theses — see Section 12.11).

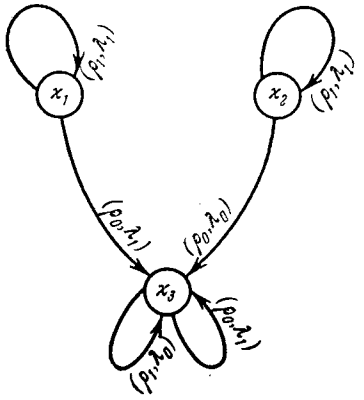


Fig. 9.2.

Let L be an arbitrary recursive set of input sequences, and let x_i and x_j be arbitrary states of s-machines S and G, respectively. Then the following theorem is true:

Theorem. The problem of recognition of equivalence of states x_i and x_j in terms of an arbitrary, effectively defined set L is algorithmically unsolvable.

We shall prove this theorem by demonstrating the algorithmic unsolvability of the narrower problem of recognition of equivalence of states in a special machine, whose allowed set L belongs to a special subclass of recursive sets. If the problem is algorithmically unsolvable in this special case, then it is certainly unsolvable in the general case.

Consider a three-state, s-machine N and its state diagram (Fig. 9.2). In this machine $r = 2$, that is, the input alphabet consists of only two symbols $\{0, 1\}$. The output can be either λ_0 or λ_1.

Now we shall deal with a special class of recursive sets containing sequences of 0 and 1, and defined as follows: Let $\varphi(t)$ be an arbitrary general recursive function, and let the set L_φ contain only the following sequences* of 0 and 1:

$$\mathrm{sg}(\varphi(0));$$
$$\mathrm{sg}(\varphi(0)),\quad \mathrm{sg}(\varphi(1));$$
$$\mathrm{sg}(\varphi(0)),\quad \mathrm{sg}(\varphi(1)),\quad \mathrm{sg}(\varphi(2));$$
$$\mathrm{sg}(\varphi(0)),\quad \mathrm{sg}(\varphi(1)),\quad \mathrm{sg}(\varphi(2)),\quad \mathrm{sg}(\varphi(3));$$
and so on.

Then set L_φ is recursive at any recursive function $\varphi(t)$. Indeed, each $(p+1)$ long sequence of 0 and 1 can be placed into correspondence with a definite value of the integer function

$$\Phi(p) = \sum_{t=0}^{t=p} \mathrm{sg}(\varphi(t))\, 2^t + 2^{p+1}$$

*The notation $\mathrm{sg}(x)$ denotes a function which is equal to 1 for $x \geqslant 1$ and equal to 0 for $x = 0$. The function is undefined for $x < 0$.

defined on set $\Lambda(L_\varphi)$. Function $\Phi(p)$ is an increasing function and, by virtue of the recursivity of $\varphi(t)$, is also recursive.* Consequently, set $\Lambda(L_\varphi)$ is consecutively enumerated as the recursive function $(d)\Phi$ increases, and therefore is a recursive set. Hence the set L_φ must also be recursive.

It is readily seen that the states $\varkappa_i$ and $\varkappa_j$ (where $i, j = 1, 2, 3$; $i \neq j$) of the machine N (Fig. 9.2) are equivalent to each other in terms of L_φ if, and only if, L_φ contains no sequences comprising ρ_0. Thus the problem of recognition of equivalence of states of N is algorithmically solvable only if there exists an algorithm capable of recognizing whether L_φ contains even one sequence comprising ρ_0. But such an algorithm cannot be written unless there is an algorithm for recognizing whether a given arbitrary recursive function $\varphi(t)$ becomes zero at some $t = t_*$. And it has been proved [142] that no such algorithm exists. For that reason, our narrow recognition problem is algorithmically unsolvable, and the generalized problem of recognition equivalence of two states of an arbitrary s-machine with respect to an arbitrary recursive set L is *a fortiori* algorithmically unsolvable. This proves the theorem.**

9.3. RECOGNITION OF THE EQUIVALENCE OF STATES IN THE CASE OF AN UNRESTRICTED SET OF INPUT SEQUENCES

Let no restrictions be imposed on the set of allowable input sequences, that is, let $L = E$. In this case the algorithm merely recognizes the simple equivalence of the s-machine states. For this case we have a straightforward and convenient algorithm, which is due to Aufenkamp and Hohn.***

To start with, let us point out an obvious attribute of equivalence of states: if any two states of an s-machine are equivalent with respect to set L_1, then they will also be equivalent with respect to set L_2, provided $L_2 \subseteq L_1$.**** Conversely, two states can be equivalent with respect to L_1, where $L_1 \supseteq L_2$, only if they are also equivalent with respect to L_2.

---

*See Sections 12.6 and 12.13.

**But in no way implies that the problem is unsolvable in special cases.

***Aufenkamp and Hohn [6] proved only the sufficiency of this algorithm. We shall give a somewhat different proof for its sufficiency, and shall also prove its necessity.

****This is read as L_1 is a subset of L_1.

Now assume that we are given some s-machine, for which we write the interconnection matrix, obtaining, for example, matrix C

$$C = \begin{array}{c} \\ \varkappa_1 \\ \varkappa_2 \\ \varkappa_3 \\ \varkappa_4 \\ \varkappa_5 \\ \varkappa_6 \end{array} \begin{array}{c} \begin{array}{cccccc} \varkappa_1 & \varkappa_2 & \varkappa_3 & \varkappa_4 & \varkappa_5 & \varkappa_6 \end{array} \\ \left[\begin{array}{cccccc} (\rho_0, \lambda_1) & (\rho_1, \lambda_2) & 0 & 0 & 0 & (\rho_2, \lambda_1) \\ (\rho_2, \lambda_0) & (\rho_1, \lambda_1) & (\rho_0, \lambda_2) & 0 & 0 & 0 \\ 0 & (\rho_0, \lambda_1) & (\rho_2, \lambda_1) & 0 & (\rho_1, \lambda_2) & 0 \\ 0 & (\rho_2, \lambda_1) & 0 & (\rho_0, \lambda_1) & (\rho_1, \lambda_2) & 0 \\ 0 & 0 & (\rho_0, \lambda_2) & (\rho_2, \lambda_0) & (\rho_1, \lambda_1) & 0 \\ 0 & (\rho_1, \lambda_1) & (\rho_0, \lambda_2) & (\rho_2, \lambda_0) & 0 & 0 \end{array}\right] \end{array}$$

We decompose this matrix into groups of rows containing identical symbol pairs. Thus rows $\varkappa_1$, $\varkappa_4$, and $\varkappa_3$ fall into group 1, and rows $\varkappa_2$, $\varkappa_5$, and $\varkappa_6$ into group 2. We rewrite matrix C so as to be able to reflect this grouping with a minimum amount of effort, and transpose the columns in the same way. We draw a horizontal line between the groups, and obtain matrix

$$C = \begin{array}{c} \\ \varkappa_1 \\ \varkappa_4 \\ \varkappa_3 \\ \varkappa_2 \\ \varkappa_5 \\ \varkappa_6 \end{array} \begin{array}{c} \begin{array}{cccccc} \varkappa_1 & \varkappa_4 & \varkappa_3 & \varkappa_2 & \varkappa_5 & \varkappa_6 \end{array} \\ \left[\begin{array}{cccccc} (\rho_0, \lambda_1) & 0 & 0 & (\rho_1, \lambda_2) & 0 & (\rho_2, \lambda_1) \\ 0 & (\rho_0, \lambda_1) & 0 & (\rho_2, \lambda_1) & (\rho_1, \lambda_2) & 0 \\ 0 & 0 & (\rho_2, \lambda_1) & (\rho_0, \lambda_1) & (\rho_1, \lambda_2) & 0 \\ \hline (\rho_2, \lambda_0) & 0 & (\rho_0, \lambda_2) & (\rho_1, \lambda_1) & 0 & 0 \\ 0 & (\rho_2, \lambda_0) & (\rho_0, \lambda_2) & 0 & (\rho_1, \lambda_1) & 0 \\ 0 & (\rho_2, \lambda_0) & (\rho_0, \lambda_2) & (\rho_1, \lambda_1) & 0 & 0 \end{array}\right] \end{array},$$

which differs from the original matrix C in that rows and columns $\varkappa_2$ and $\varkappa_4$ are transposed.

Note that the states of each group are equivalent in terms of the set L^1 comprising all allowable input sequence of unit length. Indeed, within each group the output is independent of the state of the s-machine. For example, an input ρ_0 will always produce an output λ_1 regardless of whether the machine is in state $\varkappa_1$, $\varkappa_3$, or $\varkappa_4$. Furthermore, only states belonging to the same group can be equivalent at arbitrary allowable input sequences. States belonging to different groups are *a priori* nonequivalent in such cases because they are nonequivalent even in terms of set L^1 containing sequences of only unit length.

Our grouping into submatrices is helpful in clarifying some of the equivalence relations in the matrix, but is not sufficient. It does not guarantee that two states of a group will not become nonequivalent during later operation of the machine. To elucidate all of the possible equivalence relationships, we introduce a further decomposition of our matrix into symmetrical submatrices. Thus if the previously drawn horizontal line separated the kth and the $(k+1)$th

rows, we now draw a vertical line to separate the kth and the $(k+1)$th columns. In our example such a symmetrical decomposition is obtained by drawing a vertical line to separate columns $\varkappa_3$ and $\varkappa_2$.

$$C = \begin{array}{c|ccc|ccc} & \varkappa_1 & \varkappa_4 & \varkappa_3 & \varkappa_2 & \varkappa_5 & \varkappa_6 \\ \varkappa_1 & (\rho_0, \lambda_1) & 0 & 0 & (\rho_1, \lambda_2) & 0 & (\rho_2, \lambda_1) \\ \varkappa_4 & 0 & (\rho_0, \lambda_1) & 0 & (\rho_2, \lambda_1) & (\rho_1, \lambda_2) & 0 \\ \varkappa_3 & 0 & 0 & (\rho_2, \lambda_1) & (\rho_0, \lambda_1) & (\rho_1, \lambda_2) & 0 \\ \hline \varkappa_2 & (\rho_2, \lambda_0) & 0 & (\rho_0, \lambda_2) & (\rho_1, \lambda_1) & 0 & 0 \\ \varkappa_5 & 0 & (\rho_2, \lambda_0) & (\rho_0, \lambda_2) & 0 & (\rho_1, \lambda_1) & 0 \\ \varkappa_6 & 0 & (\rho_2, \lambda_0) & (\rho_0, \lambda_2) & (\rho_1, \lambda_1) & 0 & 0 \end{array}.$$

A submatrix in which any pair of symbols present in any row is also present in all the other rows is called a 1-matrix. In our example, the two submatrices below the horizontal line are 1-matrices, but the other two are not (for example, the top left-hand submatrix has a pair (ρ_2, λ_1) in the third row not present in the other rows). We shall try to decompose them symmetrically into 1-matrices. Such a decomposition is achieved by first drawing the minimum number of horizontal lines sufficient to convert the entire matrix into 1-matrices. But this partitioning will not be symmetrical. For this reason, one also draws vertical lines between the columns corresponding to the rows already separated by the horizontal lines. After this, we check whether all resulting submatrices are 1-matrices. If not, we again draw horizontal lines, and so on, until we obtain a completely symmetrical decomposition consisting only of 1-matrices.

In our example we draw a horizontal between rows $\varkappa_4$ and $\varkappa_3$. All the resulting submatrices are 1-matrices, and the vertical, drawn between columns $\varkappa_4$ and $\varkappa_3$ to achieve symmetry, does not upset this property:

$$C = \begin{array}{c|cc|cc|cc} & \varkappa_1 & \varkappa_4 & \varkappa_3 & \varkappa_2 & \varkappa_5 & \varkappa_6 \\ \varkappa_1 & (\rho_0, \lambda_1) & 0 & 0 & (\rho_1, \lambda_2) & 0 & (\rho_2, \lambda_1) \\ \varkappa_4 & 0 & (\rho_0, \lambda_1) & 0 & (\rho_2, \lambda_1) & (\rho_1, \lambda_2) & 0 \\ \hline \varkappa_3 & 0 & 0 & (\rho_2, \lambda_1) & (\rho_0, \lambda_1) & (\rho_1, \lambda_2) & 0 \\ \hline \varkappa_2 & (\rho_2, \lambda_0) & 0 & (\rho_0, \lambda_2) & (\rho_1, \lambda_1) & 0 & 0 \\ \varkappa_5 & 0 & (\rho_2, \lambda_0) & (\rho_0, \lambda_2) & 0 & (\rho_1, \lambda_1) & 0 \\ \varkappa_6 & 0 & (\rho_2, \lambda_0) & (\rho_0, \lambda_2) & (\rho_1, \lambda_1) & 0 & 0 \end{array}.$$

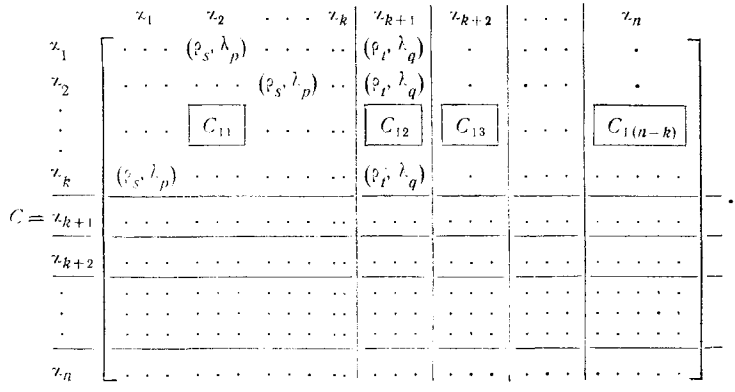
In the general case, this symmetrical decomposition will produce, after a finite number of steps, either of the following two situations:

1. The decomposition is *trivial*, that is, all the resulting submatrices are of the 1×1 order (there are separating lines between all the rows and all the columns).

2. The decomposition is *nontrivial*, that is, we have at least one submatrix of the order $m \times n$, where $\max(m, n) > 1$. In conrast to the trivial case, this partitioning gives *groups* of states. For example, the matrix shown above is split into three groups $\{\varkappa_1, \varkappa_4\}$, $\{\varkappa_3\}$, $\{\varkappa_2, \varkappa_5, \varkappa_6\}$.

The Aufenkamp-Hohn Theorem. The states of an s-machine are equivalent if, and only if, they are members of the same group formed by symmetrical decomposition of the given matrix C.

Proof of sufficiency of the conditions of the theorem. Suppose matrix C can be symmetrically and nontrivially decomposed into 1-matrices, and consider first the simple case when the matrix is



Here, we have only one group containing more than one state. This is the group of k states $\{\varkappa_1, \varkappa_2, \ldots, \varkappa_k\}$, that is, the $k \times k$ 1-matrix C_{11}. Let us write out all the input symbols appearing in C_{11}, and let these symbols be $\rho_{\alpha_1}, \rho_{\alpha_2}, \ldots, \rho_{\alpha_i}$, where $i \leqslant r$. The symbols with subscripts α_{i+1} to α_r do not appear in C_{11}. Now we shall prove that states $\varkappa_1, \varkappa_2, \ldots, \varkappa_k$ are equivalent.

Assume we have an arbitrary input sequence

$$\rho^0 \rho^1 \ldots \rho^j \rho^{j+1} \ldots \tag{9.1}$$

and that $\rho^{j+1} = \rho_{\alpha_{i+m}}$ is the first input symbol of this sequence which does not occur in C_{11}. With this input sequence, the machine output is independent of its initial state (which, by our assumption, must be one of the states of group $\{\varkappa_1, \varkappa_2, \ldots, \varkappa_k\}$). Indeed, until time j the input symbol ρ must belong in C_{11}, and therefore the machine must assume one of the states $\{\varkappa_1, \varkappa_2, \ldots, \varkappa_k\}$. But as long as the machine assumes one of these states, its output will depend only on the input (since C_{11} is a 1-matrix in which all the member pairs are identical.

Therefore, until time j the output sequence does not depend on which of the states $\{\varkappa_1, \varkappa_2, \ldots, \varkappa_k\}$ is the initial state of the machine. Then, at time $j+1$, the input signal becomes ρ^{j+1}. This input signal always shifts the machine into the same state $\varkappa_l$, regardless of which of the states $\{\varkappa_1, \varkappa_2, \ldots, \varkappa_k\}$ the machine happens to be in. This is because $C_{12}, C_{13}, \ldots, C_{1(n-k)}$ are all 1-matrices. Thus, the output is again independent of the previous state of the machine. The subsequent output of the machine is governed by the fact that at time $j+1$ it is in the state $\varkappa_l$ and, accordingly, it ceases to depend on the initial conditions.

Since sequence (9.1) was chosen at random, states $\varkappa_1, \varkappa_2, \ldots, \varkappa_k$ are equivalent.

Now consider the general case. Let C be symmetrically decomposed into 1-matrices

$$C = \begin{array}{c|c|c|c|c|c} & \varkappa_1 \ \varkappa_2 \ldots \varkappa_k & \varkappa_{k+1} \ \varkappa_{k+2} \ldots \varkappa_{k+s} & \ldots & \varkappa_{n-1} & \varkappa_n \\ \hline \varkappa_1 \ \varkappa_2 \ldots \varkappa_k & C_{11} & C_{12} & \ldots & \ldots & \ldots \\ \hline \varkappa_{k+1} \ \varkappa_{k+2} \ldots \varkappa_{k+s} & C_{21} & C_{22} & \ldots & \ldots & \ldots \\ \hline \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \hline \varkappa_{n-1} & \ldots & \ldots & \ldots & \ldots & \ldots \\ \hline \varkappa_n & \ldots & \ldots & \ldots & \ldots & \ldots \end{array}$$

We shall prove that the states occurring in any one group are equivalent. Indeed, any input signal can produce one of the following effects: (1) it can shift the machine from a state belonging to some group into a state belonging to the same group; in this case the output depends only on the input, and not on the state the machine is in; (2) it can shift the machine from a state of the ith group into a state of the jth group $(i \neq j)$. However, it is seen from the structure of matrix C, that in this case this input signal also shifts all the other states of group i into the states of group j. And once the machine is in a state of group j, its output again does not depend on the specific state in which it happens to be in; (3) it can shift the machine from a state of group i into a state $\varkappa_l$ not occurring in any group. If that happens, the input will also shift all the other states of group i into

state $\varkappa_l$, and the subsequent output of the machine will depend only on $\varkappa_l$.

The above reasoning holds for any of the groups of the matrix. Therefore, the output of the s-machine is always independent of the specific initial state of group i in which the machine happens to be. This being the case, groups of equivalent states behave as if each group were a single state. This proves the sufficiency of the conditions of the theorem.

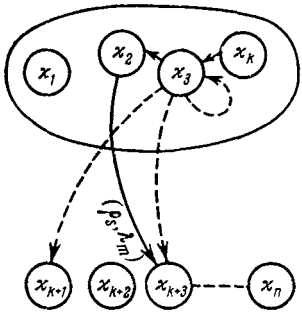


Fig. 9.3.

Proof of necessity of the conditions of the theorem. Consider first a simple case. Let the states $\varkappa_1, \varkappa_2, \ldots, \varkappa_k$ form a group of equivalent states of the s-machine, and let states $\varkappa_{k+1}, \varkappa_{k+2}, \ldots, \varkappa_n$ be nonequivalent to each other and to any state of the above group. Let us draw the state diagram of this machine. We shall now show that if state $\varkappa_i$ of group $\{\varkappa_1, \varkappa_2, \ldots, \varkappa_k\}$ is connected to any state $\varkappa_{k+m}$ outside that group by a path labeled $\rho_s \ldots$, then all the other states of that group must also be connected to the same state $\varkappa_{k+m}$, and their connecting paths must also be labeled $\rho_s \ldots$. That is, the first symbols of the path labels must coincide. For example, if state $\varkappa_2$ of Fig. 9.3 is connected to $\varkappa_{k+3}$ by a path whose label includes ρ_s as the first symbol, than the other states of the group of which $\varkappa_2$ is a member (that is, $\varkappa_1, \varkappa_3, \varkappa_4, \ldots, \varkappa_k$) must also be connected to $\varkappa_{k+3}$, and the connecting paths must also carry ρ_s as the first symbol of their labels. To prove this statement, consider state $\varkappa_2$ of the same group as $\varkappa_3$. On the face of it, the path labeled $\rho_s \ldots$, and originating in $\varkappa_3$ could follow one of the following courses:

1) it could lead from $\varkappa_3$ to one of the states of group $\{\varkappa_1, \varkappa_2, \ldots, \varkappa_k\}$;

2) it could lead from $\varkappa_3$ to one of the states $\varkappa_{k+i}$ (where $i \neq 3$), for example, to state $\varkappa_{k+1}$;

3) it could lead to state $\varkappa_{k+3}$, that is, to the same state as the path labeled $\rho_s \ldots$, originating in $\varkappa_3$.

We shall now show that case (3) is the only one possible. Indeed, assume for a moment that case (1) is possible. That would mean that the same input ρ_s could cause the machine to shift from state $\varkappa_2$ to state $\varkappa_{k+3}$, and from state $\varkappa_3$, which is equivalent to $\varkappa_2$, to some state $\varkappa_j$ of group $\{\varkappa_1, \varkappa, \ldots, \varkappa_k\}$. But state $\varkappa_{k+3}$ is by definition nonequivalent to any of the states of group $\{\varkappa_1, \varkappa_2, \ldots, \varkappa_k\}$; consequently, $\varkappa_{k+3}$ is also nonequivalent to $\varkappa_j$, and thus there would exist a sequence $\rho_{\alpha_1} \rho_{\alpha_2} \rho_{\alpha_3} \ldots$, such that its input to the machine would cause the latter

to generate different output sequences, depending on which of the states—$\varkappa_j$ or $\varkappa_{k+3}$—is the initial state of the machine. However, if this were the case, then the input sequence $\rho_s \rho_{\alpha_1} \rho_{\alpha_2} \rho_{\alpha_3} \ldots$ would also cause the generation of differing output sequences, depending on whether the initial state of the machine is $\varkappa_2$ or $\varkappa_3$. But that would be contrary to the assumed equivalence of states $\varkappa_2$ and $\varkappa_3$. Thus, case (1) is impossible.

Now assume that case (2) holds. Then an input ρ_s would shift the machine from state $\varkappa_2$ to state $\varkappa_{k+3}$, and from state $\varkappa_3$ to state $\varkappa_{k+1}$. But $\varkappa_{k+1}$ is, by definition, not equivalent to $\varkappa_{k+3}$; consequently, there would exist, just as in case (1), a sequence $\rho_{\beta_1} \rho_{\beta_2} \rho_{\beta_3} \ldots$, such that its input to the machine would cause the latter to generate different outputs, depending on which of the states—$\varkappa_{k+1}$ or $\varkappa_{k+3}$—is the initial one. But then the input of sequence $\rho_s \rho_{\beta_1} \rho_{\beta_2} \rho_{\beta_3} \ldots$ would again prove the nonequivalence of states $\varkappa_2$ and $\varkappa_3$, which would contradict the starting assumptions. Consequently, case (2) cannot hold, and the only possible case is (3), shown in Fig. 9.4. Here all states of group $\{\varkappa_1, \varkappa_2, \ldots, \varkappa_k\}$ are connected to the same state $\varkappa_{k+3}$ outside the group, and all the connecting paths bear a label whose first symbol is ρ_s. The second symbol of the label must also coincide, since otherwise it would be possible to prove by means of an input signal of length 1 that some pair of states from group $\{\varkappa_1, \varkappa_2, \ldots, \varkappa_k\}$ is nonequivalent, which would contradict the conditions of the problem.

It follows that if any state of group $\{\varkappa_1, \varkappa_2, \ldots, \varkappa_k\}$ *is connected with one of the states* $\varkappa_{k+1}, \varkappa_{k+2}, \ldots, \varkappa_n$ *by a path labeled* (ρ_s, λ_m), *then all the states of group* $\{\varkappa_1, \varkappa_2, \ldots, \varkappa_k\}$ *are also connected to that state by paths labeled* (ρ_s, λ_m) [*see Fig. 9.4*].

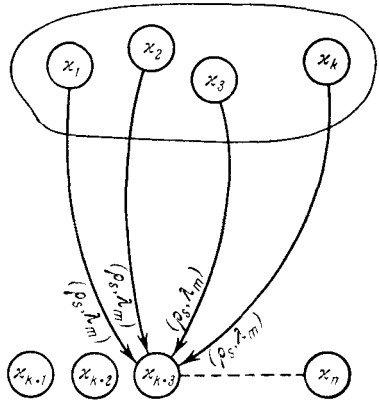


Fig. 9.4.

It also follows from the above that *if a path labeled* (ρ_s, λ_m) *connects a state of group* $\{\varkappa_1, \varkappa_2, \ldots, \varkappa_k\}$ *with another state of that group, then all the other similarly labeled paths from all the other states* $\{\varkappa_1, \varkappa_2, \ldots, \varkappa_k\}$ *must also terminate in states belonging to that group. That is, no path labeled* (ρ_s, λ_m) *leads to a state* $\varkappa_{k+1}, \varkappa_{k+2}, \ldots, \varkappa_n$ *outside the group.*

If this is so, then the interconnection matrix of our machine will be

$$C = \begin{array}{c|c|c|c|c|c} & \varkappa_1\ \varkappa_2 \ldots \varkappa_k & \varkappa_{k+1} & \varkappa_{k+2} & & \varkappa_n \\ \varkappa_1 \\ \varkappa_2 \\ \vdots \\ \varkappa_k & C_{11} & C_{12} & C_{13} & & C_{1\,(n-k+1)} \\ \hline \varkappa_{k+1} & C_{21} & \ldots & \ldots & \ldots & \ldots \\ \hline \varkappa_{k+2} & C_{31} & \ldots & \ldots & \ldots & \ldots \\ \hline \vdots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \hline \varkappa_n & C_{(n-k+1)\,1} & \ldots & \ldots & \ldots & \ldots \end{array}.$$

Here all the submatrices $C_{11}, C_{12}, \ldots, C_{1(n-k+1)}$ are 1-matrices by virtue of the above italicized statements. The other submatrices C_{ij} (where $i = 2, 3, \ldots, n-k+1$ and $j = 1, 2, \ldots, n-k+1$) are also 1-matrices since they all contain only one row (or one member).

Our arguments also hold in the general case where there are several groups of pairwise equivalent states. However, some of the individual states $\varkappa_{k+1}, \varkappa_{k+2}, \ldots, \varkappa_n$ must then be replaced by groups of states; each of these groups behaves in a manner completely analogous to the group $\{\varkappa_1, \varkappa_2, \ldots, \varkappa_k\}$ of our particular case. This concludes the proof of the theorem.

The Aufenkamp and Hohn theorem results in a simple and very convenient algorithm for determining which groups of states of a given s-machine are equivalent. This algorithm merely consists of symmetrical decomposition of the interconnection matrix of the given s-machine.

9.4 RECOGNITION OF EQUIVALENCE OF STATES FOR THE CASE OF INPUT SEQUENCES OF LIMITED LENGTH

We shall now consider the problem of equivalence of states when the set L of allowable input sequences cannot contain sequences

comprising more than q symbols (that is, we analyze the operation of the s-machine during the first q discrete instants after the input). It is required to find an algorithm recognizing those states which are equivalent in terms of L, and to group these states together.

Since the total number of differing input symbols ρ_i is finite, and since no sequence can contain more than q such symbols, the number of different sequences in set L must be finite. This being the case, the required algorithm must exist. To ascertain that any two states $\varkappa_i$ and $\varkappa_j$ of a machine are equivalent in terms of L, it is sufficient to prove that given identical inputs from L, the machine starting from state $\varkappa_i$ will generate the same output as the same machine starting from state $\varkappa_j$, and that this will happen at all possible inputs from L. One can prove this by scanning either the state diagram of the machine, its interconnection matrix, or any other of its representations, or by an experiment on an existing machine. The algorithm for recognizing equivalence thus entails scanning of all the input-output relationships which are possible for a given set of two states. This obviously is a huge task. One way of organizing and, possibly, minimizing this unwieldy scanning procedure is to raise the interconnection matrix of the given s-machine to a power, a procedure described in Section 3.6. Let us now recall the properties of matrix C^q.

1. The element C^q_{ij} of C^q enumerates all those input sequences of length q which shift the machine from state $\varkappa_i$ to state $\varkappa_j$, as well as the corresponding output sequences.

2. Since the state of the machine at $t = p + 1$ is uniquely defined by its state and input at $t = p$, a single row of C^q cannot contain two elements whose terms comprise identical input sequences.

3. Each input sequence of length q must appear in each row of C^q.

Starting from these properties of C^q, one can derive the following method for determining the states equivalent in terms of L. Let us arrange set L in order of increasing sequence length. We now take the shortest sequence of L (if there are several such sequences, all of equal length, we can use any one of these), and find in matrix C^q (where q is the maximum length of a sequence of L) all those input sequences whose initial segments coincide with our shortest sequence. We mark these coinciding segments in some way, for instance, by placing dots over each of their constituent symbols. We repeat this procedure with each successive sequence of L (sequences of equal length can be taken in an arbitrary order). Each symbol of C^q is marked only once; that is, if we find a matching sequence in

C^q, we place dots only over those symbols which are still unmarked. This matching procedure finally gives a matrix which has sequences carrying dots over all their symbols as well as sequences that have only some initial segments marked, or no markings at all.

For example, if $q = 3$; the set L contains the four sequences

$$\rho_1;\ \rho_1\rho_2;\ \rho_2\rho_1;\ \rho_2\rho_1\rho_2,$$

and the s-machine has the state diagram of Fig. 3.11 (for its matrix C^3 see Section 3.6), then the matrix sequences are marked as follows:

$$C^3 = \begin{array}{c|ccc} & \alpha_1 & \alpha_2 & \alpha_3 \\ \alpha_1 & [(\dot{\rho}_1\rho_1\dot{\rho}_2,\ \lambda_3\lambda_1\lambda_3) \vee & [(\dot{\rho}_1\dot{\rho}_2\rho_2,\ \lambda_3\lambda_2\lambda_2) \vee & [(\dot{\rho}_1\dot{\rho}_2\rho_1,\ \lambda_3\lambda_2\lambda_1) \vee \\ & \vee (\dot{\rho}_2\dot{\rho}_1\dot{\rho}_2,\ \lambda_1\lambda_1\lambda_3)] & \vee (\rho_2\dot{\rho}_2\dot{\rho}_2,\ \lambda_1\lambda_2\lambda_2) \vee & \vee (\rho_2\rho_2\rho_1,\ \lambda_1\lambda_2\lambda_1)] \\ & & \vee (\dot{\rho}_1\rho_1\rho_1,\ \lambda_3\lambda_1\lambda_2) \vee & \\ & & \vee (\dot{\rho}_2\dot{\rho}_1\rho_1,\ \lambda_1\lambda_1\lambda_2)] & \\ \alpha_2 & (\rho_2\dot{\rho}_1\dot{\rho}_2,\ \lambda_2\lambda_1\lambda_3) & [(\dot{\rho}_1\dot{\rho}_2\rho_1,\ \lambda_1\lambda_3\lambda_3) \vee & [(\rho_2\rho_2\rho_1,\ \lambda_2\lambda_2\lambda_1) \vee \\ & & \vee (\dot{\rho}_1\dot{\rho}_2\rho_2,\ \lambda_1\lambda_3\lambda_1) \vee & \vee (\dot{\rho}_1\rho_1\rho_1,\ \lambda_1\lambda_2\lambda_1)] \\ & & \vee (\rho_2\rho_2\rho_2,\ \lambda_2\lambda_2\lambda_2) \vee & \\ & & \vee (\dot{\rho}_1\rho_1\rho_2,\ \lambda_1\lambda_2\lambda_2) \vee & \\ & & \vee (\dot{\rho}_2\dot{\rho}_1\rho_1,\ \lambda_2\lambda_1\lambda_2)] & \\ \alpha_3 & (\dot{\rho}_1\rho_1\rho_2,\ \lambda_2\lambda_1\lambda_3) & [(\dot{\rho}_2\dot{\rho}_1\dot{\rho}_2,\ \lambda_3\lambda_3\lambda_2) \vee & [(\dot{\rho}_2\dot{\rho}_1\rho_1,\ \lambda_3\lambda_3\lambda_1) \vee \\ & & \vee (\rho_2\dot{\rho}_2\rho_2,\ \lambda_3\lambda_1\lambda_3) \vee & \vee (\rho_2\rho_2\rho_1,\ \lambda_3\lambda_1\lambda_1) \vee \\ & & \vee (\dot{\rho}_1\dot{\rho}_2\rho_2,\ \lambda_2\lambda_2\lambda_2) \vee & \vee (\dot{\rho}_1\dot{\rho}_2\rho_1,\ \lambda_2\lambda_2\lambda_1)] \\ & & \vee (\dot{\rho}_1\rho_1\rho_1,\ \lambda_2\lambda_1\lambda_2)] & \end{array}$$

After marking, we delete from C^q all those input sequences (together with the corresponding outputs) which do not carry dots (in our example of C^3, these are the sequences $\rho_2\rho_2\rho_2$ and $\rho_2\rho_2\rho_1$). In input sequences where dots appear only over the initial segments, we delete the unmarked symbols, that is, the tail ends. We also chop off the corresponding tail-end sections of the output sequences, and we obtain a *C^q matrix abridged by L.* For example, our C^3 matrix is abridged by L to give

$$\begin{array}{c|ccc} & \alpha_1 & \alpha_2 & \alpha_3 \\ \alpha_1 & (\rho_1,\ \lambda_3) \vee (\rho_2\rho_1\rho_2,\ \lambda_1\lambda_1\lambda_3) & (\rho_1,\ \lambda_3) \vee (\rho_2\rho_1,\ \lambda_1\lambda_1) \vee & (\rho_1\rho_2,\ \lambda_3\lambda_2) \\ & & \vee (\rho_1\rho_2,\ \lambda_3\lambda_2) & \\ \alpha_2 & (\rho_2\rho_1\rho_2,\ \lambda_2\lambda_1\lambda_3) & (\rho_1\rho_2,\ \lambda_1\lambda_3) \vee (\rho_1,\ \lambda_1) \vee & (\rho_1,\ \lambda_1) \\ & & \vee (\rho_2\rho_1,\ \lambda_2\lambda_1) & \\ \alpha_3 & (\rho_1,\ \lambda_2) & (\rho_2\rho_1\rho_2,\ \lambda_3\lambda_3\lambda_2) \vee & (\rho_2\rho_1,\ \lambda_3\lambda_3) \vee \\ & & \vee (\rho_1\rho_2,\ \lambda_2\lambda_2) \vee (\rho_1,\ \lambda_2) & \vee (\rho_1\rho_2,\ \lambda_2\lambda_2) \end{array}$$

Thus the abridged matrix contains only those input sequences (and the corresponding outputs) which are present in L. Now we can

define a simple scheme for recognition of equivalence of states: *Two states x_i and x_j of an s-machine are equivalent in terms of L if, and only if, the pairs of input and output sequences of row i of its abridged (by L) C^q matrix match exactly those of row j and there are no unmatched pairs in either row.* Thus, in the abridged matrix of our example there are no two rows with exactly the same pairs, and therefore this s-machine has no states equivalent to each other in terms of L.

However, even this algorithm, which is an improvement over the disorganized scanning of all possible input-output relationships, is still unwieldy, especially at large values of q. For this reason, one tries to avoid the necessity of scanning all input sequences from L. Instead, one tries to reduce each problem to those particular cases where such scanning is not needed. Let us consider one such case.

Let set L contain all the sequences of length smaller or equal to q. Set L is a subset of set E containing all input sequences. For that reason, any two states equivalent in terms of E (that is, simply equivalent) are also equivalent in terms of L. Now we have to ask ourselves when do groups of states of a given machine, which are equivalent in terms of E, coincide with the groups equivalent in terms of L; that is, when are the states which are equivalent in terms of L also equivalent in terms of E? If these two decompositions into groups coincide, then we can use the Aufenkamp - Hohn algorithm; however, if the groupings do not coincide, we may have to resort to the scanning procedure described above, or to some new method.

The answer to this question is associated with the relationship between the number of states of the machine k, and the maximum length of an allowable input sequence q. We shall show that if q is sufficiently large then it may be possible to recognize all the *non-equivalent* states. Then each pair of states nonequivalent in terms of E will also be nonequivalent in terms of L.

Assume that we are given a sequential machine S with k states, and that we symmetrically decompose its interconnection matrix by means of the Aufenkamp - Hohn method. Now we have k^* groups of equivalent states (obviously, $k^* \leqslant k$).

We shall try to prove that if $q \geqslant k^* - 1$, then the grouping of equivalent states, obtained by the Aufenkamp - Hohn procedure, produces groups which coincide with those equivalent in terms of L; if that is true, then at $q \geqslant k^* - 1$ we can solve the equivalence problem by means of the Aufenkamp - Hohn algorithm, and the number of resulting groups will indeed be equal to k^*.

Let us devise a machine S^* having k^* states and the following characteristics: (a) for each state of machine S there is an equivalent

state of machine S^* and, conversely, for each state of machine S^* there is an equivalent state of machine S; (b) no two states of S^* are equivalent. It will be shown in Section 9.7 that such a machine can always be devised.

We shall now apply Moore's theorem (Section 11.2) which states that if a machine N has k states and all the states are nonequivalent to each other, then for each pair of states $\varkappa_i$ and $\varkappa_j$ there always exists an input sequence not longer than $k-1$ that allows us to differentiate between these two states. Since all the states of S^* are pairwise nonequivalent [see characteristic (b) above], sequences not longer than k^*-1 will differentiate between all the nonequivalent states of this machine. Therefore, if $q \geqslant k^*-1$, all these "differentiating" sequences are contained in L, all states nonequivalent in terms of L can be distinguished, and the Aufenkamp - Hohn algorithm can be used.

However, if $q < k^*-1$, then the grouping in terms of L may not coincide with the grouping with respect to E. In this case one may be forced to resort to the scanning procedure in order to obtain a grouping in terms of L (one way of accomplishing such scanning is the above method of raising matrix C to the power of q).

Sometimes one can avoid the scanning in such cases by estimating the lower bound of the number of states equivalent in terms of L. Thus let us partition matrix C into 1-matrices using only horizontal lines. Then the states of the machine are divided into m groups. These will be groups of states equivalent with respect to set L^1 of all the input sequences of length 1 (set L^1 coincides with the alphabet $\{\rho_1, \rho_2, \ldots, \rho_r\}$). Obviously, the number of groups of states equivalent with respect to L cannot be less than m, since $q \geqslant 1$ and $L^1 \subseteq L$ and, consequently, any two states equivalent with respect to L are also equivalent with respect to L^1. Thus, m is the desired lower bound.

For the same reason, k^* is the upper bound of the number of groups of states which are equivalent in terms of L, since $L \subseteq E$, so that any two simply equivalent states are also equivalent in terms of L.

Thus, if m turns out to be equal to k^* then, despite the fact that $q < k^*-1$, one can use the Aufenkamp - Hohn algorithm.

In the practical application of the Aufenkamp - Hohn algorithm, m and k^* are obtained at different stages of the computation. Thus m is obtained in the first stage, when horizontal lines are drawn to partition matrix C into groups. If, however, the vertical lines drawn subsequently to achieve symmetry "spoil" this grouping, then other horizontal lines must be drawn, and so on, so that ultimately one

one obtains $k^* > m$. Thus one knows immediately whether the Aufenkamp - Hohn algorithm is applicable.

Restricting our discussion of the equivalence problem to the cases described in this and the previous sections, we shall make two brief observations regarding other definitions of the allowable input sequences L.

1. One important case (particularly in the theory of relay-contact circuits) is that where L contains all sequences in which no two identical symbols are repeated consecutively. It can be shown that for this case there exists an algorithm for recognizing equivalent states. However, the present authors know of no algorithm which would be suitable for practical use.

2. If Aufenkamp constraints are operative, then the very statement of the problem must be changed: in this case it makes no sense to talk of two equivalent states $\varkappa_i$ and $\varkappa_j$ since states $\varkappa_i$ and $\varkappa_j$ may allow different sets of input sequences. Here $L_{\varkappa_i}$ may be forbidden in $\varkappa_j$, and vice versa. However, in this case one may sometimes use a concept which is akin to that of equivalence. This is the concept of compatibility of states, which is defined as follows:

Two states—state $\varkappa_i$ of machine S and state ζ_j of machine G—are said to be compatible if, and only if both machines—machine S in initial state $\varkappa_i$ and machine G in initial state ζ_j—having acquired any input sequence from the intersection of set L_{κ_i} with set L_{ζ_j}, will generate identical output sequences (in particular, S and G may be the same machine).* In accordance with this definition, states $\varkappa_i$ and ζ_j must be compatible if that intersection is an empty set, that is, if states $\varkappa_i$ and ζ_j have no allowable input sequences in common. If L_{κ_i} and L_{ζ_j} coincide then, of course, compatibility reduces to equivalence in terms of the common set.

Now, the group of states $\{\varkappa_1, \varkappa_2, \ldots, \varkappa_k\}$ is said to be a *group of pseudoequivalent states* if, and only if, any two states $\varkappa_i$ and $\varkappa_j$ of that group are compatible. This concept is frequently very useful; in particular, it can be applied for minimization of an s-machine which is subject to Aufenkamp constraints (see Section 9.8).

9.5. EQUIVALENCE, MAPPING AND MINIMIZATION OF SEQUENTIAL MACHINES

So far, we discussed the equivalence of individual states; now we shall turn to the equivalence of entire s-machines.

---

*An *intersection* of two sets contains all points belonging to both sets.

Two s-machines, S and G, are said to be equivalent in terms of L if, and only if, for each state $\varkappa_i$ of S there exists at least one state ζ_j of G equivalent to it in terms of L and, conversely, if for each state ζ_s of G there exists at least one state $\varkappa_t$ of S equivalent to it with respect to L.

This definition says any input sequence from L must be allowed both in S and in G. If the set of all sequences allowed in S is L_S, and the analogous set for G is L_G, then L must satisfy the condition

$$L \subseteq L_S \cap L_G,$$

where $L_S \cap L_G$ denotes the intersection of sets L_S and L_G. When $L = E$ (that is, L contains all the possible sequences), we shall say that the machines S and G are ***simply equivalent.*** In this case $L_S = L_G = L = E$.

Machine S maps onto machine G in terms of set L (or G maps S in terms of L) if, and only if, for each state $\varkappa_i$ of S there exists at least one state ζ_j of G equivalent to it in terms of L. If $L = E$, then S ***simply maps*** onto G.

From our definitions of mapping and equivalence we can deduce the following: if machine S maps onto machine G in terms of L, and G maps onto S in terms of the same L, then S and G are machines which are equivalent in terms of L.

The equivalence relationship between S and G is denoted by $S \sim G$, while the mapping of S onto G is written as $S \subset G$.

Equivalent machines are identical as far as processing of input sequences into output sequences is concerned. If machine S maps (or maps in terms of L) onto machine G, then this means that G substitutes for S (however, the converse is not true).

Consider two equivalent s-machines S_1 and S_2, and let their states be partitioned into groups of equivalent states. Now we take some such group s_1^α of S_1 and select any state $\varkappa_i$ from this group. Then S_2 will have a state $\varkappa_j$ equivalent to $\varkappa_i$. Let $\varkappa_j$ belong to the group (of equivalent states) s_2^β of S_2. If that is so, then any state belonging to s_1^α of S_1 is equivalent to any state belonging to s_2^β of S_2. However, none of the states of s_1^α of S_1 is equivalent to any of the states of s_2^γ of S_2, if $\gamma \neq \beta$. Therefore, each group of equivalent states of S_1 corresponds to one and only one group of equivalent states of S_2. The symmetry of the equivalence relationship (it follows from $S_1 \sim S_2$ that $S_2 \sim S_1$), implies that the converse statement is also true, that is, each group of equivalent states of S_2 corresponds to one, and only one group of equivalent states of S_1. Accordingly, the two equivalent machines S_1 and S_2 contain the same number

of groups of equivalent states, and machines S_1 and S_2 differ only in the number of states in each of the corresponding equivalent groups.

If, however, we are given two machines S and G such that G maps S ($S \subset G$), there is no one-to-one correspondence between their groups of equivalent states: all we can say is that to each group of equivalent states of S there corresponds one and only one group of equivalent states of G. However, the converse is not true. Accordingly, G may have more groups of equivalent states than S; thus, machines S and G may differ not only in the number of states in each (equivalent) group, but also in the number of (equivalent) groups.

All of the above also holds if we consider equivalence and mapping in relation to a set L restricted *per se*.

Now let us discuss the minimization of an s-machine S. Minimization of an s-machine S with respect to a set L (of allowable sequences) shall mean finding another s-machine G satisfying these two conditions:

1) G maps S ($G \supset S$) with respect to L.

2) There is no other s-machine mapping S in terms of L and containing fewer states than G.

An s-machine G satisfying these conditions is said to be *minimal for S in terms of L*.

Let us point out that if there exists an algorithm for recognizing states equivalent in terms of L then, in principle, there also exists a trivial minimization algorithm in terms of L. Indeed, if machine S has k states, then the number of internal states in G (which is minimal for S) cannot exceed k. In principle, therefore, we could scan all the machines whose number of states does not exceed k (there is a finite number of such machines). And since there must exist an algorithm for recognizing states equivalent with respect to L, we can check whether each of these machines maps S. Obviously, such a trivial algorithm has no practical value, and we would like to find practical algorithms. So far, such an algorithm exists only for the case where all input sequences are allowed. We shall describe it in the next section.

9.6. MINIMIZATION OF A SEQUENTIAL MACHINE WITH AN UNRESTRICTED SET OF ALLOWABLE INPUT SEQUENCES

Let S be a sequential machine with k internal states decomposed into groups of equivalent states as in Section 9.3. (Figure 9.5 shows a section of the state diagram of this machine.) Consider the first of these groups. Its states are the termini of paths from other states.

In turn, as we have shown in Section 9.3, any state of Group 1 may also be the origin of either of the following paths:

a) a path leading to another state of Group 1. If the first symbol in the label of this path is ρ_p, then all similarly labeled paths, originating in any state of Group 1, must also terminate at a state of this group. The second symbols in the labels of all these paths are identical.

b) a path leading to a state of another group, for example, state $\varkappa_j$ of Group M. If the first symbol in the label of this path is ρ_s, then all similarly labeled paths, originating in any state of Group 1, must also terminate in $\varkappa_j$. The second symbols in the labels of all these paths are identical.

Because they exhibit these characteristics, we can replace all the states of Group 1 by a single state. All the paths to the individual states now terminate in the circle replacing that group. The paths originating in the states of this group will, in case (a), be replaced by a loop labeled (ρ_p, λ_q), and, in the case (b) by a path labeled (ρ_s, λ_t) originating in the new circle and leading to a circle replacing the states of Group M. Figure 9.6 shows such a replacement for the partial state diagram of Fig. 9.5.

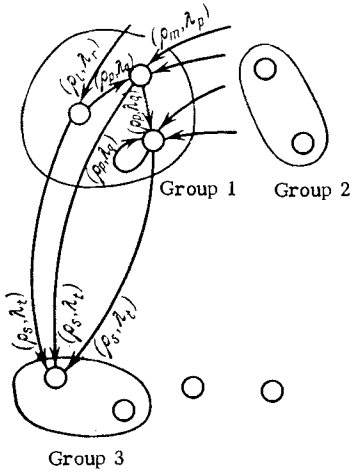


Fig. 9.5.

In the same way, we replace all the other groups of states. As a result, the machine S is transformed into machine G. It is evident that G is equivalent to S.

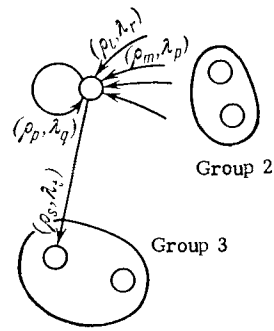


Fig. 9.6.

Indeed, by virtue of characteristics (a) and (b), the output of S at any input sequence and in any initial state $\varkappa_i$ is identical to that which would be generated by G at the same input sequence, provided G was in the initial state $\varkappa_j$ which replaces the group of which $\varkappa_i$ was a member. At the same time, the number of states of machine G is

equal to the number of groups of equivalent states of S, and one cannot further reduce this number by combining these states into groups.

It has been shown in Section 9.5 that all equivalent machines have the same number of equivalent groups, and that the number of such groups in machines mapping such equivalent machines cannot be lower. Thus a minimal machine cannot have fewer states than there are groups of equivalent states in the machine being minimized. *For this reason, machine G is, indeed, minimal for S.* It follows from this that, *in the absence of bounds on the set of input sequences*, (a) the minimal *machine belongs to the class of equivalent machines*, and (b) *the minimization problem is merely one of finding groups of equivalent states, that is, it can be solved by means of the Aufenkamp - Hohn algorithm* (see Section 9.3).

Since this algorithm is used, it is convenient to work with matrix C rather than the state diagram, and replace groups of states by a single one directly in the matrix. For example, consider the matrix C of Section 9.3

$$C = \begin{array}{c|cccccc} & \varkappa_1 & \varkappa_2 & \varkappa_3 & \varkappa_4 & \varkappa_5 & \varkappa_6 \\ \varkappa_1 & (\rho_0, \lambda_1) & (\rho_1, \lambda_2) & 0 & 0 & 0 & (\rho_2, \lambda_1) \\ \varkappa_2 & (\rho_2, \lambda_0) & (\rho_1, \lambda_1) & (\rho_0, \lambda_2) & 0 & 0 & 0 \\ \varkappa_3 & 0 & (\rho_0, \lambda_1) & (\rho_2, \lambda_1) & 0 & (\rho_1, \lambda_2) & 0 \\ \varkappa_4 & 0 & (\rho_2, \lambda_1) & 0 & (\rho_0, \lambda_1) & (\rho_1, \lambda_2) & 0 \\ \varkappa_5 & 0 & 0 & (\rho_0, \lambda_2) & (\rho_2, \lambda_0) & (\rho_1, \lambda_1) & 0 \\ \varkappa_6 & 0 & (\rho_1, \lambda_1) & (\rho_0, \lambda_2) & (\rho_2, \lambda_0) & 0 & 0 \end{array}.$$

By symmetrical decomposition into 1-matrices we obtain

$$C = \begin{array}{c|cc|c|ccc} & \varkappa_1 & \varkappa_4 & \varkappa_3 & \varkappa_2 & \varkappa_5 & \varkappa_6 \\ \varkappa_1 & (\rho_0, \lambda_1) & 0 & 0 & (\rho_1, \lambda_2) & 0 & (\rho_2, \lambda_1) \\ \varkappa_4 & 0 & (\rho_0, \lambda_1) & 0 & (\rho_2, \lambda_1) & (\rho_1, \lambda_2) & 0 \\ \hline \varkappa_3 & 0 & 0 & (\rho_2, \lambda_1) & (\rho_0, \lambda_1) & (\rho_1, \lambda_2) & 0 \\ \hline \varkappa_2 & (\rho_2, \lambda_0) & 0 & (\rho_0, \lambda_2) & (\rho_1, \lambda_1) & 0 & 0 \\ \varkappa_5 & 0 & (\rho_2, \lambda_0) & (\rho_0, \lambda_2) & 0 & (\rho_1, \lambda_1) & 0 \\ \varkappa_6 & 0 & (\rho_2, \lambda_0) & (\rho_0, \lambda_2) & (\rho_1, \lambda_1) & 0 & 0 \end{array}.$$

Thus we have three groups of equivalent states, $\{\varkappa_1, \varkappa_4\}$, $\{\varkappa_3\}$, and $\{\varkappa_2, \varkappa_5, \varkappa_6\}$.

The replacement of groups of states of a state diagram by a single state is equivalent to the replacement of each 1-matrix of the symmetrical decomposition by a single element, which is a disjunction (union) of all the elements of the 1-matrix being replaced.

In our example, this replacement will give the following interconnection matrix for the minimal machine G:

$$C = \begin{array}{c} \\ \varkappa_1' \\ \varkappa_2' \\ \varkappa_3' \end{array} \begin{array}{c} \begin{array}{ccc} \varkappa_1' & \varkappa_2' & \varkappa_3' \end{array} \\ \begin{bmatrix} (\rho_0, \lambda_1) & 0 & (\rho_1, \lambda_2) \vee (\rho_2, \lambda_1) \\ 0 & (\rho_2, \lambda_1) & (\rho_0, \lambda_1) \vee (\rho_1, \lambda_2) \\ (\rho_2, \lambda_0) & (\rho_0, \lambda_2) & (\rho_1, \lambda_1) \end{bmatrix} \end{array}$$

Its state diagram is shown in Fig. 9.7.

So far we have dealt with the minimization of an s-machine whose set of input sequences is infinite. The problem of minimization of an s-machine in which *per se* restrictions are operative is tied to the still unsatisfactorily solved problem of finding groups of states equivalent in terms of L for the same case (see Section 1 and Section 9.4).

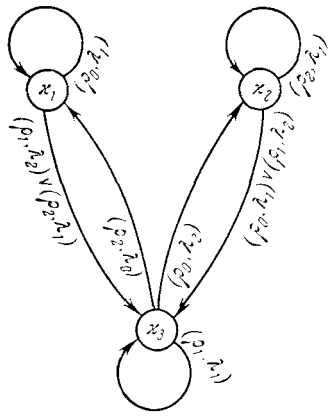


Fig. 9.7.

In addition, minimization with respect to $L \neq E$ is associated with the following additional difficulty, which would exist even if we had an algorithm for finding groups of states equivalent in terms of L. Thus, earlier in this section we were able to replace a group of states by a single state by using properties of the paths in the state diagram (see p. 238). However, if $L \neq E$, then, generally speaking, the paths do not possess the properties specified in (a) and (b), p. 238. Thus two paths, the labels of which contain identical first symbols and originating in states which are equivalent in terms of L may terminate in states which are nonequivalent in terms of L. Consider, for example, the section of the state diagram (Fig. 9.8) for the case where L does not contain any sequence with two consecutive identical symbols. Let states κ_1 and κ_2 be equivalent in terms of L, that is, belong to one group. Further, let states κ_3 and κ_4 be equivalent in terms of L' which contains all the sequences of L except those beginning with the symbol ρ_s, and let κ_3 and κ_4 be nonequivalent in terms of L since they generate different output symbols at those sequences from L which begin with ρ_s. Then the paths of Fig. 9.8 do not contradict the equivalence of κ_1 and κ_2 in terms of L (for L does not contain any sequences beginning with two consecutive symbols ρ_s), but they do contradict condition (b) of p. 238.

It follows from the foregoing that at $L \neq E$ the states belonging to one group of equivalent states cannot, generally speaking, be replaced by a single state. If this were done, two paths labeled with an identical first symbol ρ_s would originate in the same new state and lead to two different states—a condition which contradicts the very definition of an s-machine.

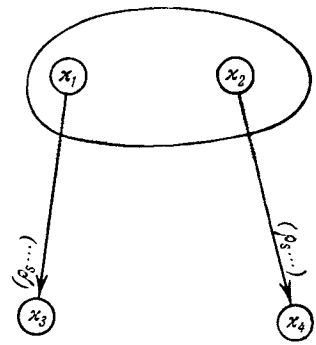


Fig. 9.8.

So far we have not imposed any restrictions on the processing of sequences by the s-machine which is being minimized. However, in the next section we shall consider a special case where the s-machine operates as a finite automaton.

9.7. MINIMIZATION OF A SEQUENTIAL MACHINE WHEN IT OPERATES AS A FINITE AUTOMATON

Assume that we are given the basic table of a finite automaton whose states are coded in symbols from the alphabet $\{\varkappa_1, \varkappa_2, \ldots, \varkappa_h\}$. Let us recode the symbols, replacing all $\varkappa_i$ by λ_i; we shall assume that $l = k$, where l is the number of symbols λ. Obviously, the basic table of the finite automaton now contains λ's with the subscripts of the $\varkappa$'s they replace.

It is required to devise a minimal sequential machine which would realize this automaton, that is, would process inputs into outputs in the same way as the automaton. The set of allowable input sequences may be restricted or unrestricted.

We shall consider two cases: a case where there are no restrictions on the input sequences, and a case where the input sequences may not contain two consecutive identical symbols.

Case 1. Set L Contains All Possible Sequences (that is, $L = E$)

We shall analyze this case on an automaton A given in the form of Table 9.1.

We have shown in Section 9.6 that, in the absence of restrictions on the input sequences, the minimal s-machine for a given machine N belongs to the class of machines equivalent to N, and that none of

its states has other equivalents. Consequently, our required minimal s-machine must also be equivalent to automaton A.

Table 9.1

λ \ ρ	ρ_1	ρ_2	ρ_3
λ_1	λ_4	λ_3	λ_2
λ_2	λ_1	λ_2	λ_4
λ_3	λ_2	λ_1	λ_1
λ_4	λ_4	λ_3	λ_2

Let us first construct an s-machine which is not minimal, but which is suitable for further minimization. It will have as many states as there are rows in the automaton of Table 9.1, and its state diagram (Fig. 9.9) has as many circles, numbered consecutively 1 - 4. We draw paths between these circles as per the Table 9.1. Each path has a label whose first symbol is the subscript of the corresponding ρ from the table, while the second symbol is the number of the circle in which the path terminates. Since this is a diagram of some s-machine, we replace the numbers in each circle by symbols $\varkappa_i$ and the numbers (m, n) in the paths

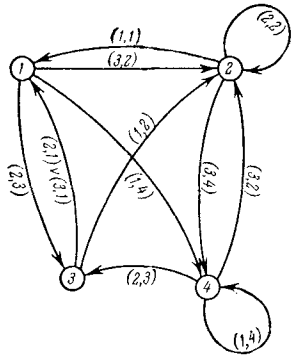


Fig. 9.9.

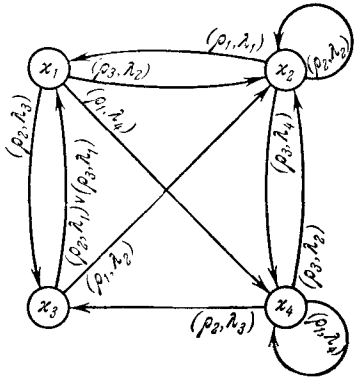


Fig. 9.10.

labeled by symbols (ρ_m, λ_n); this gives Fig. 9.10. Now we derive the interconnection matrix of this machine:

$$C = \begin{array}{c} \\ \varkappa_1 \\ \varkappa_2 \\ \varkappa_3 \\ \varkappa_4 \end{array} \begin{array}{c} \begin{array}{cccc} \varkappa_1 & \varkappa_2 & \varkappa_3 & \varkappa_4 \end{array} \\ \begin{bmatrix} 0 & (\rho_3, \lambda_2) & (\rho_2, \lambda_3) & (\rho_1, \lambda_4) \\ (\rho_1, \lambda_1) & (\rho_2, \lambda_2) & 0 & (\rho_3, \lambda_4) \\ (\rho_2, \lambda_1) \vee (\rho_3, \lambda_1) & (\rho_1, \lambda_2) & 0 & 0 \\ 0 & (\rho_3, \lambda_2) & (\rho_2, \lambda_3) & (\rho_1, \lambda_4) \end{bmatrix} \end{array}.$$

By analogy with the above state diagram, all the nonzero elements of the interconnection matrix belonging to the same column have identical subscripts on the λ symbol. These subscripts coincide

with the number of the column. This matrix C can be transformed into the interconnection matrix C' of an equivalent minimal s-machine, which therefore is a minimal s-machine operating in the same way as automaton A.

First we decompose C into 1-matrices by means of horizontals only. We get a 1-matrix from rows 1 and 4. We transpose these rows and get

$$C = \begin{array}{c|cccc} & \varkappa_1 & \varkappa_4 & \varkappa_2 & \varkappa_3 \\ \varkappa_1 & 0 & (\rho_1, \lambda_4) & (\rho_3, \lambda_2) & (\rho_2, \lambda_3) \\ \varkappa_4 & 0 & (\rho_1, \lambda_4) & (\rho_3, \lambda_2) & (\rho_2, \lambda_3) \\ \hline \varkappa_2 & (\rho_1, \lambda_1) & (\rho_3, \lambda_4) & (\rho_2, \lambda_2) & 0 \\ \hline \varkappa_3 & (\rho_2, \lambda_1) \vee (\rho_3, \lambda_1) & 0 & (\rho_1, \lambda_2) & 0 \end{array}.$$

Now we draw horizontals between rows κ_4 and κ_2, and between rows κ_2 and κ_3, and obtain three 1-matrices whose columns contain either zeros or identical pairs [for example, the two-row matrix on top has only zeros in column 1, only pairs (ρ_1, λ_4) in column 2, and so on]. This is the result of the previously mentioned property of C: the second digits are the same in each column of C. But in 1-matrices, where the columns contain only identical pairs, the first digits of each column will also coincide. If this is so, then all we need to do in order to form groups of equivalent states to partition C into 1-matrices by horizontals only: since the elements in the columns of each 1-matrix coincide, vertical lines cannot "spoil" this symmetrical grouping.

This property, in turn, means the following: the groups of states of an s-machine (with matrix C) which are simply equivalent, and those which are equivalent in terms of set L_1 comprising all input sequences of length 1, coincide. Therefore, to find all the groups of equivalent states, it is sufficient to partition C into groups of states equivalent in terms of L_1, a partition achieved simply by decomposing C into 1-matrices by means of horizontal lines.

In our example, state $\varkappa_1$ of matrix C is equivalent to state $\varkappa_4$. To construct a minimal s-machine, we replace these two states with a single state $\varkappa_1$. Then we draw verticals between columns 2 and 3 and 3 and 4 of C. In this symmetrical decomposition we replace each newly generated 1-matrix by the union

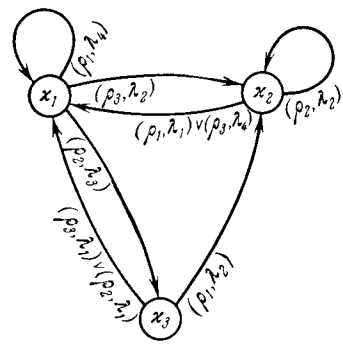


Fig. 9.11.

of all its elements, and obtain the interconnection matrix C' of the minimal s-machine:

$$C' = \begin{array}{c} \\ \varkappa_1 \\ \varkappa_2 \\ \varkappa_3 \end{array} \begin{array}{c} \begin{array}{ccc} \varkappa_1 & \varkappa_2 & \varkappa_3 \end{array} \\ \begin{bmatrix} (\rho_1, \lambda_4) & (\rho_3, \lambda_2) & (\rho_2, \lambda_3) \\ (\rho_1, \lambda_1) \vee (\rho_3, \lambda_4) & (\rho_2, \lambda_2) & 0 \\ (\rho_2, \lambda_1) \vee (\rho_3, \lambda_1) & (\rho_1, \lambda_2) & 0 \end{bmatrix} \end{array}$$

The corresponding state diagram is shown in Fig. 9.11.

Note also that for each set of identical rows of matrix C (in our case, rows 1 and 4) there always exists a set of identical rows in the automaton table (here, rows 1 and 4 of Table 9.1), and vice versa. Consequently, inspection of the automaton table immediately shows the number of states of a minimal s-machine realizing this automaton (one needs only to count the number of differing rows in the table of the automaton).

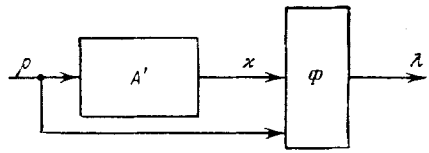


Fig. 9.12.

Having the state diagram, we can compile the table of the automaton A' and converter Φ which comprise the minimal s-machine operating as automaton A in accordance with Fig. 9.12. Our state diagram (Fig. 9.10) thus yields Tables 9.2 and 9.3.

Table 9.2

$\varkappa$ \ ρ	ρ_1	ρ_2	ρ_3
$\varkappa_1$	$\varkappa_1$	$\varkappa_3$	$\varkappa_2$
$\varkappa_2$	$\varkappa_1$	$\varkappa_2$	$\varkappa_1$
$\varkappa_3$	$\varkappa_2$	$\varkappa_1$	$\varkappa_1$

Table 9.3

$\varkappa$ \ ρ	ρ_1	ρ_2	ρ_3
$\varkappa_1$	λ_4	λ_3	λ_2
$\varkappa_2$	λ_1	λ_2	λ_4
$\varkappa_3$	λ_2	λ_1	λ_1

Again, the table of automaton A' (Table 9.2) could have been obtained directly from the table of automaton A (Table 9.1) merely by deleting one of the identical rows (the fourth; if there are several such rows,

all but one are deleted), and then replacing throughout the remainder of the table those symbols which are the same as the heading(s) of the deleted row(s) [λ_4 in our example] by the heading of the retained row (in our case, we replace λ_4 by λ_1).

The converter table can also be obtained directly from the automaton table. Again we delete superfluous identical rows of Table 9.1 (row 4), and in the remaining table substitute $\varkappa_i$'s for λ_i's in all row headings.

Thus we have a simple, straightforward algorithm for direct derivation of the tables of automaton A' and converter Φ which, in accordance with the scheme of Fig. 9.12, constitute the minimal s-machine realizing automaton A. The state diagram and the interconnection matrix were only necessary for proving the validity of this algorithm.

Case 2. Set L Has No Sequences Comprising Two Consecutive Identical Symbols

If L_1 is the set of input sequences of length 1, and E is the set containing all possible input sequences, then obviously we shall have the following relationship:

$$L_1 \subset L \subset E. \tag{9.2}$$

If the number of groups of equivalent states is m^*, the numbers of groups of states equivalent in terms of L and L_1 are, respectively, m and m^{**}, then by virtue of (9.2)

$$m^{**} \leqslant m \leqslant m^*.$$

In Case 1 we have shown that the groupings of equivalent states and of states equivalent in terms of L_1 coincide. Consequently, $m^{**} = m^*$, and from (9.2) we get

$$m^{**} = m = m^*.$$

For this reason states equivalent in terms of L_1 will also be equivalent in terms of E in this case. Therefore one can minimize the numbers of states by replacing each group by a single state, using the above method where it was assumed that $L = E$. Thus minimal s-machines for sets E and L coincide in Case 2, and the minimization proceeds as if there were no restrictions on the input sequences.

9.8. MINIMIZATION OF MACHINES IN THE CASE OF AUFENKAMP-TYPE CONSTRAINTS

The obvious approach to the minimization problem in this case is as follows.

Let N be an s-machine with k states, subject to arbitrary Aufenkamp-type constraints. We shall say that to minimize N means devising a new machine P with a minimal number of states such that for each state $\varkappa_i$ of N there is at least one state χ_j of P. States $\varkappa_j$ must satisfy the following conditions:

a) Any input sequence allowed in $\varkappa_i$ of N is allowed in χ_j of P.

b) If N is in state $\varkappa_i$ and P is in state χ_j, and if some arbitrary sequence from the set of input sequences allowed in N when in the state $\varkappa_i$ is fed to both machines, then both will convert it into identical output sequences.

We shall say that a *machine* P (which need not necessarily be minimal) *satisfying conditions (a) and (b) realizes a pseudomapping of machine* N. Thus P "can do" whatever N can. That is, it can take any input sequence allowed in N and process it into the same output sequence.

We shall now describe an Aufenkamp algorithm resulting in a machine P which is a pseudomapping of machine N and has fewer states than N, but is not necessarily minimal.

On a state diagram, the presence of Aufenkamp-type constraints manifest itself in that the number of paths originating at some circles is smaller than that of various inputs $\rho_1, \rho_2, \ldots, \rho_r$. This means that the machine cannot respond to some inputs when it is in certain states.

The state diagram, in turn, is the starting point for the construction of the interconnection matrix. Again, the effect of the constraints on that matrix is that the latter may contain rows with fewer symbol pairs than there are inputs $\rho_1, \rho_2, \ldots, \rho_r$. For example, consider the matrix

$$C = \begin{array}{c} \\ \varkappa_1 \\ \varkappa_2 \\ \varkappa_3 \\ \varkappa_4 \\ \varkappa_5 \end{array} \begin{array}{c} \begin{array}{ccccc} \varkappa_1 & \varkappa_2 & \varkappa_3 & \varkappa_4 & \varkappa_5 \end{array} \\ \begin{bmatrix} 0 & (\rho_1, \lambda_0) & (\rho_3, \lambda_2) & 0 & 0 \\ (\rho_2, \lambda_1) & 0 & 0 & (\rho_3, \lambda_2) & 0 \\ 0 & (\rho_3, \lambda_2) & 0 & 0 & (\rho_1, \lambda_0) \\ 0 & 0 & (\rho_2, \lambda_2) & 0 & 0 \\ (\rho_2, \lambda_2) & 0 & (\rho_1, \lambda_3) & 0 & 0 \end{bmatrix} \end{array}.$$

The first row lacks a symbol pair incorporating the input ρ_2, the second lacks the pair associated with ρ_1, and the fourth contains only a single pair (that associated with ρ_2).

We shall say that a submatrix of an interconnection matrix C is a generalized 1-matrix if it has the following property: *if any row of the generalized 1-matrix contains a pair* (ρ_m, λ_n), *then none of the remaining rows of that matrix will contain pairs in which this input symbol* ρ_m *is associated with a different symbol* λ.

We shall cite here, without proof, the following theorem of Aufenkamp [5]: *Assume the interconnection matrix C is decomposed by horizontal lines into groups of rows constituting generalized 1-matrices, and is then further partitioned by vertical lines to achieve a symmetrical decomposition into generalized 1-matrices. Provided no two generalized 1-matrices of a given group contain the same input symbol* ρ_m, *the states of this group are pseudoequivalent.* Thus, machine N can be minimized by replacing each group of pseudoequivalent states by a single state. This is done by replacing each generalized 1-matrix of a symmetrical decomposition by one term which represents a union (disjunction) of all the elements of the 1-matrix being replaced. This gives a matrix C' of machine P which realizes a pseudomapping of N and has fewer states provided, of course, that the symmetrical decomposition of C is nontrivial.

To illustrate, let N have the state diagram of Fig. 9.13, with matrix C shown above (p. 246). First, we draw horizontals to decompose C into generalized 1-matrices. In contrast to the case where there were no restrictions and there was only one way of partitioning C into 1-matrices, now we have several possibilities. For example, the rows of C may be divided into three groups, the first comprising the rows 1 and 4, the second — rows 2 and 3, and the third—row 5. However, we can also divide C into two groups, the first comprising 1, 2 and 3, and the second—rows 4 and 5. It is important to realize that the mode of partitioning will definitely affect the possibility of minimizing N. For example, let us partition C in the most economic way, that is, into the two groups discussed above. We thus draw a horizontal between rows 3 and 4 and obtain two generalized 1-matrices. Then, for symmetry, we draw a vertical between columns 3 and 4. This "spoils" our decomposition because the top group of rows now contains two generalized 1-matrices, each containing ρ_1, and ρ_3 in violation of the above-cited theorem of Aufenkamp. The states of the group are thus not pseudoequivalent. To remedy this situation, we draw horizontals between rows 1 and 2, and 2 and 3

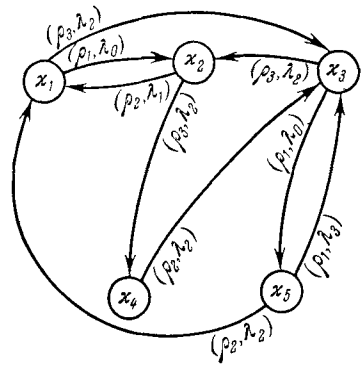


Fig. 9.13.

(in the general case, there are several possibilities for achieving such adjustments), so that we now have four groups, none of which contains two generalized 1-matrices with the same symbol ρ. But, for symmetry, we must also draw verticals between columns 1 and 2, and 2 and 3. This again spoils the decomposition because the group comprising rows 4 and 5 now has two generalized 1-matrices, each containing a pair with ρ_2. We are therefore forced to draw a horizontal between rows 4 and 5, and a corresponding vertical between the columns. Obviously this decomposition is trivial, and thus the machine cannot be minimized in this way.

Assume, however, that we start with a seemingly less economical partition of C, that is, the one comprising three groups discussed above (group 1 comprises rows 1 and 4, group 2—rows 2 and 3, and group 3—row 5). We thus rewrite C as follows:

$$C = \begin{array}{c|ccc|cc} & \kappa_1 & \kappa_4 & \kappa_2 & \kappa_3 & \kappa_5 \\ \kappa_1 & 0 & 0 & (\rho_1, \lambda_0) & (\rho_3, \lambda_2) & 0 \\ \kappa_4 & 0 & 0 & 0 & (\rho_2, \lambda_2) & 0 \\ \hline \kappa_2 & (\rho_2, \lambda_1) & (\rho_3, \lambda_2) & 0 & 0 & 0 \\ \kappa_3 & 0 & 0 & (\rho_3, \lambda_2) & 0 & (\rho_1, \lambda_0) \\ \hline \kappa_5 & (\rho_2, \lambda_2) & 0 & 0 & (\rho_1, \lambda_3) & 0 \end{array}.$$

Here we have only one conflict—two matrices of group 2 contain ρ_3. This is easily fixed by drawing a horizontal between rows $\varkappa_2$ and $\varkappa_3$, and a corresponding vertical between the columns. We now have one group of pseudoequivalent states comprising more than one row, that is, group 1. This group can be replaced by a single state, giving the matrix of a somewhat minimized machine P' (it has four states vs the five of N) which realizes the pseudomapping of N.

Finally, we could partition C into the following three groups: group 1 comprising rows 1 and 2, group 2—rows 3 and 4, and group 3—row 5. This immediately yields a symmetrical decomposition which needs no "fixing":

$$C = \begin{array}{c|cc|cc|c} & \kappa_1 & \kappa_2 & \kappa_3 & \kappa_4 & \kappa_5 \\ \kappa_1 & 0 & (\rho_1, \lambda_0) & (\rho_3, \lambda_2) & 0 & 0 \\ \kappa_2 & (\rho_2, \lambda_1) & 0 & 0 & (\rho_3, \lambda_2) & 0 \\ \hline \kappa_3 & 0 & (\rho_3, \lambda_2) & 0 & 0 & (\rho_1, \lambda_0) \\ \kappa_4 & 0 & 0 & (\rho_2, \lambda_2) & 0 & 0 \\ \hline \kappa_5 & (\rho_2, \lambda_2) & 0 & (\rho_1, \lambda_3) & 0 & 0 \end{array}.$$

As a result, the pseudomapping of N is realized by a machine P'' which has only three states and a matrix C'':

$$C'' = \begin{array}{c} \\ \kappa_1 \\ \kappa_2 \\ \kappa_3 \end{array} \begin{array}{c} \begin{array}{cccc} \kappa_1 & & \kappa_2 & \kappa_3 \end{array} \\ \begin{bmatrix} (\rho_1, \lambda_0)\ (\rho_2, \lambda_1) & (\rho_3, \lambda_2) & 0 \\ (\rho_3, \lambda_2) & (\rho_2, \lambda_2) & (\rho_1, \lambda_0) \\ (\rho_2, \lambda_2) & (\rho_1, \lambda_3) & 0 \end{bmatrix} \end{array}.$$

Its state diagram is shown in Fig. 9.14.

Thus to achieve optimum results in applying this algorithm one must try out all the possibilities for symmetrical decomposition of C into generalized 1-matrices.* Furthermore, this algorithm does not necessarily yield a minimal machine P: the matrix C of machine N may be decomposed into groups of pseudoequivalent states replaceable by a single state thereby minimizing it without fulfilling the conditions of the second Aufenkamp theorem. For example, consider the machine of Fig. 9.15. Here, state $\varkappa_1$ does not admit an input ρ_1. The corresponding interconnection matrix C is

$$C = \begin{array}{c} \\ \varkappa_1 \\ \varkappa_2 \\ \varkappa_3 \end{array} \begin{array}{c} \begin{array}{ccc} \varkappa_1 & \varkappa_2 & \varkappa_3 \end{array} \\ \begin{bmatrix} (\rho_2, \lambda_0) & (\rho_3, \lambda_0) & 0 \\ (\rho_1, \lambda_0) & (\rho_3, \lambda_0) & (\rho_2, \lambda_0) \\ (\rho_3, \lambda_0) & (\rho_2, \lambda_0) & (\rho_1, \lambda_1) \end{bmatrix} \end{array}.$$

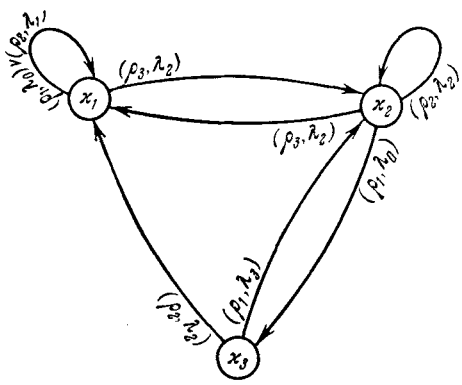


Fig. 9.14.

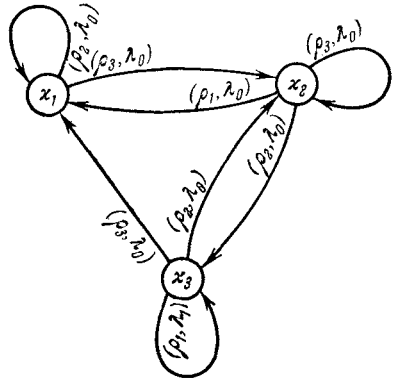


Fig. 9.15.

*Note that this algorithm is tantamount to the scanning of all the possible additional definitions of this s-machine, that is, to the scanning of all the possibilities for drawing missing paths on the state diagram, with subsequent minimization of machines so obtained without any restrictions on the input sequence.

There are only two ways in which C could be symmetrically partitioned into generalized 1-matrices:

$$C = \begin{array}{c} \\ \varkappa_1 \\ \varkappa_2 \\ \varkappa_3 \end{array} \begin{array}{c} \begin{array}{cc} \varkappa_1 & \varkappa_2 \end{array} \\ \left[\begin{array}{cc|c} (\rho_2, \lambda_0) & (\rho_3, \lambda_0) & 0 \\ (\rho_1, \lambda_0) & (\rho_3, \lambda_0) & (\rho_2, \lambda_0) \\ \hline (\rho_3, \lambda_0) & (\rho_2, \lambda_0) & (\rho_1, \lambda_1) \end{array}\right] \end{array},$$

$$C = \begin{array}{c} \\ \varkappa_1 \\ \varkappa_3 \\ \varkappa_2 \end{array} \begin{array}{c} \begin{array}{cc} \varkappa_1 & \varkappa_3 \end{array} \\ \left[\begin{array}{cc|c} (\rho_2, \lambda_0) & 0 & (\rho_3, \lambda_0) \\ (\rho_3, \lambda_0) & (\rho_1, \lambda_1) & (\rho_2, \lambda_0) \\ \hline (\rho_1, \lambda_0) & (\rho_2, \lambda_0) & (\rho_3, \lambda_0) \end{array}\right] \end{array}.$$

Obviously, neither decomposition satisfies the second Aufenkamp theorem: in the first case, the two top generalized 1-matrices contain the same pair (ρ_2, λ_0), while in the second case the common pairs are (ρ_2, λ_0) and (ρ_3, λ_0). Nevertheless this machine can be minimized, the corresponding minimal s-machine (two states) being that of Fig. 9.16. Here state $\varkappa_A$ of the pseudomapping corresponds to states $\varkappa_1$ and $\varkappa_2$ of the original machine, while state $\varkappa_B$ is equivalent to state $\varkappa_3$.*

Now we shall describe a method devised by Gill [149], which yields a minimal machine for any given s-machine subject to Aufenkamp-type constraints. This method requires, as a first step, that all pairs of compatible states of the given machine be determined. There are methods for determining the compatibility of states, but we shall describe only one.

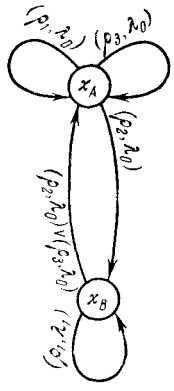


Fig. 9.16.

Suppose, for example, that we want to find out whether the ith and the jth states of a given machine are compatible. To achieve this, we construct a "tree" from the common starting point (i, j). Its branches correspond to the inputs which are common to states $\varkappa_i$ and $\varkappa_j$. If this procedure yields different outputs even for a single input, then we immediately know that states $\varkappa_i$ and $\varkappa_j$ are not compatible. If, however, the states prove compatible, then we write over the branches the corresponding input-output pairs, and at their ends the pairs of states into which $\varkappa_i$ and $\varkappa_j$ are shifted by these inputs. Each pair of such states then serves as the starting node for another branch of the tree. During

*This example also shows that no additional definition of the machine of Fig. 9.15 yields a minimal machine with two states: for no path containing ρ_1 in the label and originating at $\varkappa_1$ will generate equivalent states, regardless of what λ is in the label.

this construction we keep on crossing out nodes if:

1. The same node (that is, a node with a label consisting of the same symbols) has already been encountered anywhere else on the tree.

2. The node label consists of two identical digits, that is, if an input shifts both states $\varkappa_i$ and $\varkappa_j$ into the same state $\varkappa_m$.

3. No new branches can be drawn from this node, that is, the states corresponding to that node have no common inputs.

We stop either if states $\varkappa_i$ and $\varkappa_j$ are incompatible or if all the paths of the tree lead to crossed out nodes. In the latter case, we may conclude that the pairs of states corresponding to all the nodes (crossed out or otherwise) of the tree are also compatible.

For example, Fig. 9.17 illustrates the tree for states $\varkappa_1$ and $\varkappa_2$ of the machine of Fig. 9.15, while Fig. 9.19 shows the tree for states $\varkappa_2$ and $\varkappa_3$ of the machine of Fig. 9.18. In either case, the tested states prove to be compatible; in the first case, we also have another pair of compatible states ($\varkappa_1$ and $\varkappa_3$), in addition to $\varkappa_1$ and $\varkappa_2$, whereas in the second case, $\varkappa_4$ and $\varkappa_6$, $\varkappa_3$ and $\varkappa_5$, and $\varkappa_1$ and $\varkappa_2$ turn out to be compatible, in addition to $\varkappa_2$ and $\varkappa_3$.

This procedure also yields an estimate of the maximum number of steps (the worst case) needed to determine the compatibility of two states of a machine with n states. This estimate is obtained from*

$$l = \frac{n^2 - n}{2}.$$

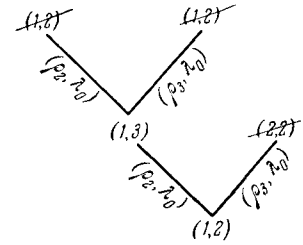


Fig. 9.17.

Now we can determine all the pairs of compatible states for any machine. The machine of Fig. 9.15 has two such pairs [$\{\varkappa_1, \varkappa_2\}$ and $\{\varkappa_1, \varkappa_3\}$], while that of Fig. 9.18 has nine $\{\varkappa_1, \varkappa_2\}$, $\{\varkappa_1, \varkappa_3\}$, $\{\varkappa_1, \varkappa_5\}$, $\{\varkappa_2, \varkappa_3\}$, $\{\varkappa_2, \varkappa_4\}$, $\{\varkappa_2, \varkappa_5\}$, $\{\varkappa_3, \varkappa_5\}$, $\{\varkappa_3, \varkappa_6\}$, $\{\varkappa_4, \varkappa_6\}$. These pairs can now be divided into groups of pseudoequivalent states. For example, the preceding list contains pairs $\{\varkappa_1, \varkappa_2\}$, $\{\varkappa_1, \varkappa_3\}$ and $\{\varkappa_2, \varkappa_3\}$, so that states $\varkappa_1$, $\varkappa_2$ and $\varkappa_3$ form a group of pseudoequivalent states $\{\varkappa_1, \varkappa_2, \varkappa_3\}$.**

Following this line of reasoning, we can divide the states of the machine into a minimal number of groups of pseudoequivalent states.

*This estimate is accurate, since there are cases where l is attained. It is interesting to compare this estimate with that for the number of steps necessary to recognize the nonequivalence of states in a machine without restrictions (see Section 11.2); in this case, $l = (n - 1)$, that is, the n^2 terms do not appear.

**Remember that a group of pseudoequivalent states is one in which all the states are pairwise compatible.

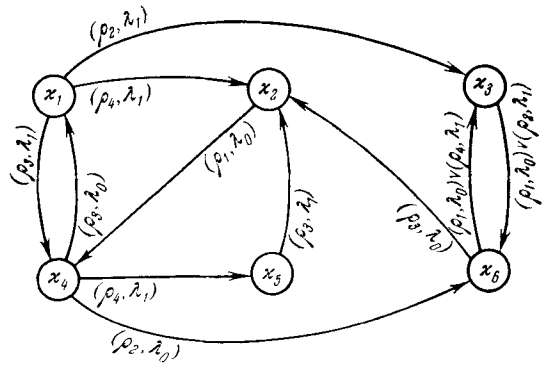


Fig. 9.18.

In our example, there will be four such groups:

$$\{x_1, x_2, x_3, x_5\}, \{x_2, x_4\}, \{x_3, x_6\}, \{x_4, x_6\}.$$

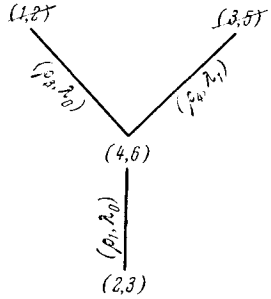


Fig. 9.19.

In the general case (just as in our example), these groups intersect.

Let us now return to the minimization of an s-machine subject to Aufenkamp-type constraints. Assume an arbitrary s-machine with n states, and assume that at least one minimal machine $S_{\min}$ with k states can be constructed for it. If $k < n$, then at least one state of $S_{\min}$ must pseudomap two or more states of S. Assume that Σ_i is the set of all states of S which correspond to state x_i of $S_{\min}$, and assume that such sets of states $\Sigma_1, \Sigma_2, \ldots, \Sigma_k$ of S can be assembled for all the states $x_1, x_2, \ldots, x_k$ of $S_{\min}$. Such grouping of states of S has the following properties:

1. The grouping $\Sigma_1, \Sigma_2, \ldots, \Sigma_k$ embraces all the states of S, that is, each state of S belongs to at least one set Σ.

2. States belonging to any one set Σ_i are pseudoequivalent.

3. All states of a given group Σ_i which allow a given input ρ_s, are shifted by it into states of the same new group Σ_j (in particular, i can be equal to j).

While the first two of these properties are obvious, the third requires an explanation. For example, let state x_i of $S_{\min}$ belong to Σ_i, and let state x_j, into which x_i is shifted by input ρ_s, belong to group Σ_j. Now assume that there exists a state of Σ_i which is shifted by ρ_s into a state x_l not belonging to Σ_j. Then observation of the

behavior of S and $S_{\min}$ at all input sequences allowed for states of Σ_i leads to the following conclusions: (a) any input allowed in state $\varkappa_l$ of S is also allowed in state $\varkappa_j$ of $S_{\min}$; and (b) if the input is a sequence allowed in state $\varkappa_l$, then both machines (that is, S starting from $\varkappa_l$, and $S_{\min}$ starting from $\varkappa_j$) will generate identical output sequences. But this simply means that state $\varkappa_l$ corresponds to state $\varkappa_j$ under pseudomapping and consequently it must, contrary to our initial assumption, belong to group Σ_j, so that the third property must hold.

The grouping $\Sigma_1, \Sigma_2, \ldots, \Sigma_k$ is known as the *specific grouping of states of machine S.*

It follows from the foregoing that the minimization algorithm involves finding a minimal specific grouping of states of S (of which there may be one or more), and the subsequent replacement of each group Σ_i by a single state. Since all states of each group are pseudoequivalent, any group Σ_i must belong to (or coincide with) some group of the minimal decomposition of the states of S into groups of pseudoequivalent states. Thus, this minimization algorithm consists of scanning of all various possible specific groupings of S in the search for the minimal one—a very cumbersome procedure. However, there are algorithms for organizing this scanning to reduce waste motions (see, for example, [149]). There are also "intermediate" algorithms which, while reducing the amount of scanning required, give better results than Aufenkamp's algorithm, even though they do not assume minimality.

Let us now return to our two examples (Figs. 9.15 and 9.18).

For the machine of Fig. 9.15, the grouping into pseudoequivalent states $\{\varkappa_1, \varkappa_2\}$ and $\{\varkappa_1, \varkappa_3\}$ is also the minimal specific grouping. We shall prove this.

Let us code group $\{\varkappa_1, \varkappa_2\}$ by A, and group $\{\varkappa_1, \varkappa_3\}$ by B. Now let us trace the possible results of various inputs:

A	ρ_1	$\varkappa_1$ does not allow ρ_1, $\varkappa_2$ shifts to $\varkappa_1$,	A or B
	ρ_2	$\varkappa_1$ shifts to $\varkappa_1$, $\varkappa_2$ shifts to $\varkappa_3$,	B
	ρ_3	$\varkappa_1$ shifts to $\varkappa_2$, $\varkappa_2$ shifts to $\varkappa_2$,	A
B	ρ_1	$\varkappa_1$ does not allow ρ_1, $\varkappa_3$ shifts to $\varkappa_3$,	B
	ρ_2	$\varkappa_1$ shifts to $\varkappa_1$, $\varkappa_3$ shifts to $\varkappa_2$,	A
	ρ_3	$\varkappa_1$ shifts to $\varkappa_2$, $\varkappa_3$ shifts to $\varkappa_1$,	A

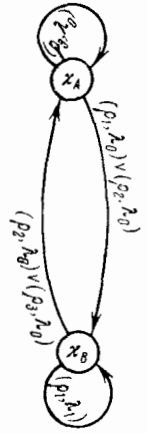


Fig. 9.20.

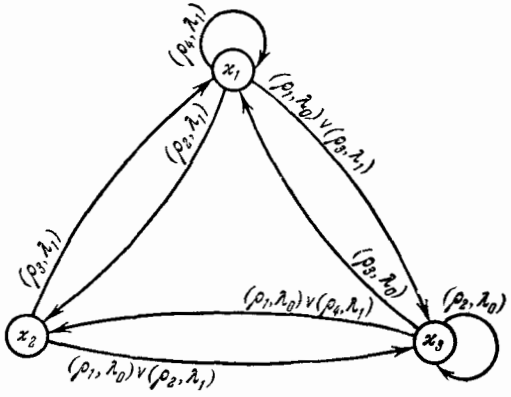


Fig. 9.21.

Now we readily construct two minimal machines for the machine of Fig. 9.15, replacing states of A by a state $\varkappa_A$ and those of B by $\varkappa_B$ (Figs. 9.16 and 9.20). For the machine of Fig. 9.18 there also are

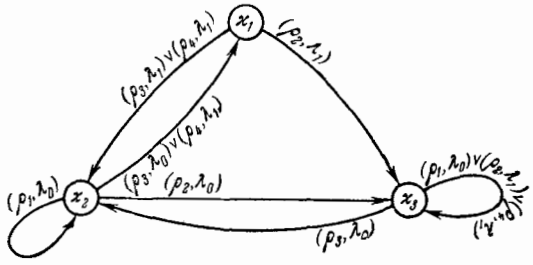


Fig. 9.22.

two possible minimal specific groupings: $\{\varkappa_1, \varkappa_2\}$, $\{\varkappa_3, \varkappa_5\}$, and $\{\varkappa_4, \varkappa_6\}$, or $\{\varkappa_1, \varkappa_5\}$, $\{\varkappa_2, \varkappa_4\}$, and $\{\varkappa_3, \varkappa_6\}$. The state diagram of the minimal machine corresponding to the first of these is shown in Fig. 9.21, while that of the second one is represented in Fig. 9.22.

9.9. ANOTHER DEFINITION OF EQUIVALENCE OF SEQUENTIAL MACHINES

Sometimes one encounters in the literature a definition of equivalence of sequential machines which differs from that of Section 9.4.

Outwardly, that definition appears similar to ours, but in reality there is a vast difference between them.

That other definition may be formulated as follows: two s-machines S and G are equivalent if at any (identical) input to both machines, there is at least one state ξ_j of G for each state $\varkappa_i$ of S, and at least one state $\varkappa_i$ of S for each state ξ_j of G such that S and G, starting from $\varkappa_i$ and ξ_j, respectively, will generate identical outputs.

In this case, the equivalence between states depends, in general, on the input sequence. At some inputs some states of S may correspond to some states of G, but at other inputs the same states of S may correspond to different states of G (and conversely). The only requirement is that there be a unique relationship between states at any one input.

From the practical point of view, the disadvantage of this definition is that in order to find an initial state equivalent to a given one, one must know beforehand the corresponding input sequence. However, in most problems of practical importance the input sequence is not known in advance.

The definition of Section 9.4 imposes more stringent requirements: the equivalence between the states of S and G should not depend on any one input, but must hold for all allowable inputs. Thus if state $\varkappa_i$ of machine S corresponds to state ξ_j of an equivalent machine G, then S and G, starting from states $\varkappa_i$ and ξ_j, respectively, must generate identical outputs at all identical inputs (provided, of course, the inputs are allowed).*

Since the above definition of equivalence is less stringent than that of Section 9.4, it should yield minimal equivalent s-machines with fewer states than those possible in terms of the definition of Section 9.4. Let us illustrate this on an example.

Table 9.4

λ \ p	p_1	p_2
λ_1	λ_3	λ_3
λ_2	λ_2	λ_3
λ_3	λ_2	λ_1

Example. We are given an automaton A and a set L of allowed input sequences; the latter consists of all sequences which do not have two consecutive identical symbols. It is required to construct a minimal s-machine mapping (in terms of L) the automaton A specified by the basic Table 9.4.** The state diagram of A is shown in Fig. 9.23. In accordance with Section 9.7, it follows from Table 9.4, that when

*In some papers equivalence in the sense of Section 9.4 is referred to as *strong* equivalence, while that defined above is called *weak*.

**In Table 9.4, $\varkappa_i$ is already replaced by λ_i (see Section 9.7).

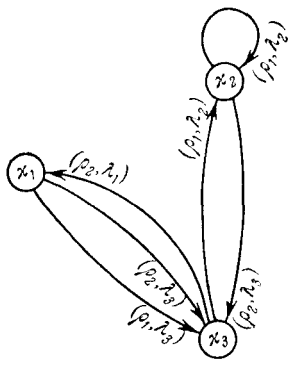


Fig. 9.23.

the definition of Section 9.4 is used, the minimal s-machine operating in the same way as A will have three states (since no two rows of Table 9.4 coincide).

Let us now use the other definition of equivalence and modify the state diagram of Fig. 9.23, replacing it with the diagram of an equivalent (in the sense of Section 9.4) machine, which is convenient for further minimization. The modification procedure is as follows: we replace each circle $\varkappa_i$ of Fig. 9.23 by r circles denoted by $\varkappa_i'$, $\varkappa_i''$, $\varkappa_i'''$, ... From each of the r new circles we draw the same paths, with the same labels, as those which originated in circle $\varkappa_i$. However, the paths terminating in circle $\varkappa_i$ will now be redirected to the new circles in the following manner: circle $\varkappa_i'$ will be the terminal of only those paths (previously leading to the circle $\varkappa_i'$) whose label contains ρ_1 as the first symbol; similarly circle $\varkappa_i''$ will be the terminal of paths whose label has ρ_2, and so on. We thus obtain the diagram of Fig. 9.24. Note that the loop at circle $\varkappa_2$ of Fig. 9.23, labeled (ρ_1, λ_2), is considered as both originating and terminating in that circle; in Fig. 9.24 it is replaced by paths originating in circles $\varkappa_2'$ and $\varkappa_2''$ and terminating only in circle $\varkappa_2''$. Circle $\varkappa_2'$ is seen to be associated with a loop path. Let us also mention that because only paths labeled λ_i lead to circle $\varkappa_i$ of Fig. 9.23 (this is because the s-machine operates as an automaton), the diagram of Fig. 9.24 has the corresponding property: a circle of group $\varkappa_i'$, $\varkappa_i''$, $\varkappa_i'''$, etc., can only be the terminal of paths labeled λ_i. In Fig. 9.24, these groups of circles are encircled with dotted lines.

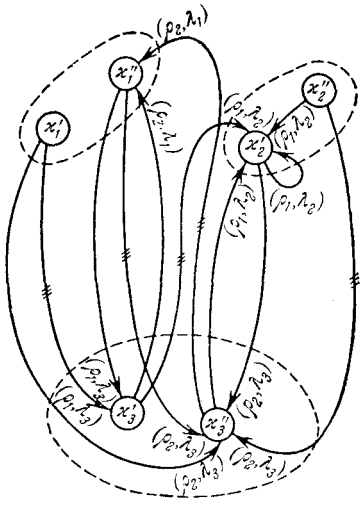


Fig. 9.24.

The modified state diagram of Fig. 9.24 is that of a machine N' which is equivalent (in the sense of Section 9.4) to A (in terms of the set E of all possible input sequences). To check whether the diagrams of Figs. 9.23 and 9.24 pertain to

equivalent machines, it is sufficient to prove that those states in each group of Fig. 9.24 which are encircled by a dotted line, are equivalent (that is, form a group of equivalent states). We replace each such group by single state and return to Fig. 9.23.

We now have machine N' which is equivalent (in the sense of Section 9.4) to A but has many more states (rn states). However, N' allows us an easy transition from *per se* constraints on the input sequences to Aufenkamp-type constraints.

Our inputs to N' shall be exclusively from L (since $L \subset E$), N' is also equivalent to A, in the sense of Section 9.4, in terms of L. Now consider some state of N', for instance, $\varkappa_1''$. We can reach $\varkappa_1''$ only via path (ρ_2, λ_1), that is, upon an input ρ_2; we can leave $\varkappa_1''$ only via path (ρ_1, λ_3), since two inputs ρ_2 cannot succeed each other (condition of problem, see p. 255). It seems, therefore, that we shall never be able to use path (ρ_2, λ_3), leaving $\varkappa_1''$, with the exception of the case in which the machine starts to work in state $\varkappa_1''$: in this case, an input ρ_2 shifts it to $\varkappa_3''$. But we can now use our new definition of equivalence to avoid this complication, for we can now substitute $\varkappa_2'$ [from which there is a path (ρ_2, λ_3) to $\varkappa_3''$] for the initial state $\varkappa_1''$, which solves our problem. Therefore, we can delete path (ρ_2, λ_3) from $\varkappa_1''$ to $\varkappa_3''$.

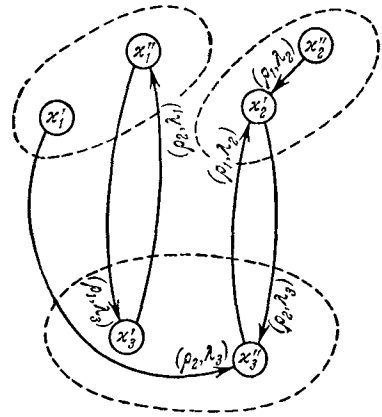


Fig. 9.25.

Following this line of reasoning, we can delete from the state diagram all the paths marked (≡). The general "algorithm for deleting path" is as follows: the path labeled ρ_s originating at $\varkappa_i^s$ (where s is the number of primes) should be deleted. The diagram of Fig. 9.24, minus the deleted paths, is shown in Fig. 9.25, and represents machine N''. That machine operates exactly as N' (and also as A), provided all the inputs belong to set L. But something has happened in this transformation, because now N'' is equivalent, in terms of L, to N' (and consequently, also to A) in a new sense: correspondence between states of N'' and N' now depends on the input sequence. Also, the diagram of N'' shows that a given circle (state) is no longer the origin of all paths that started in circles of machine A; that is, we have transformed the machine from one subject to constraints *per se* to one with Aufenkamp-type constraints.

We shall now minimize N'' by means of the algorithm of Section 9.8.

The interconnection matrix C'' of N'' is

$$C''=\begin{array}{c} \\ x_1' \\ x_1'' \\ x_2' \\ x_2'' \\ x_3' \\ x_3'' \end{array}\begin{array}{c} \begin{array}{cccccc} x_1' & x_1'' & x_2' & x_2'' & x_3' & x_3'' \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & (\rho_2, \lambda_3) \\ 0 & 0 & 0 & 0 & (\rho_1, \lambda_3) & 0 \\ 0 & 0 & 0 & 0 & 0 & (\rho_2, \lambda_3) \\ 0 & 0 & (\rho_1, \lambda_2) & 0 & 0 & 0 \\ 0 & (\rho_2, \lambda_1) & 0 & 0 & 0 & 0 \\ 0 & 0 & (\rho_1, \lambda_2) & 0 & 0 & 0 \end{bmatrix} \end{array}.$$

In general, C'' has a property which follows from the above-mentioned properties of the state diagram: the column with the heading x_i^s (where s is the number of primes) can contain only pairs (ρ_s, λ_i).

Our algorithm minimizes the number of states of the machine by symmetrical decomposition of the starting matrix into generalized 1-matrices. In the case of C'', we need only to draw horizontals to obtain generalized 1-matrices: verticals cannot "spoil" the grouping. These horizontals may be drawn in many ways. One way is to draw them between rows 2 and 3, and 4 and 5. This partitions the states of N'' into groups $\{x_1', x_1''\}$, $\{x_2', x_2''\}$ and $\{x_3', x_3''\}$. If we then replace each group with a single state, we get again the starting automaton A of Fig. 9.23, which had three states. However, a better grouping is obtained by drawing a horizontal between rows 3 and 4 of C''. This divides all the states into groups $\{x_1', x_1'', x_2'\}$ and $\{x_2'', x_3', x_3''\}$. (There is no way of obtaining fewer than two groups because rows 1 and 4 can never be part of one generalized 1-matrix.) Now, we draw a symmetrical vertical line between columns 3 and 4, combine the elements of each generalized 1-matrix, and get the interconnection matrix C''' of a minimal s-machine N'''

$$C'''=\begin{array}{c} \\ x_1 \\ x_2 \end{array}\begin{array}{c} \begin{array}{cc} x_1 & x_2 \end{array} \\ \begin{bmatrix} 0 & (\rho_1, \lambda_3)\vee(\rho_2, \lambda_3) \\ (\rho_1, \lambda_2)\vee(\rho_2, \lambda_1) & 0 \end{bmatrix} \end{array},$$

whose state diagram is shown in Fig. 9.26. Machine N''' is a pseudo-mapping of N''. But here the sets of inputs allowable in all the states of N'' coincide with each other and with L. Therefore, N'' also maps N'' in terms of set L.

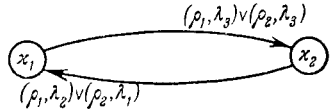


Fig. 9.26.

Note that C'' contains two pairs of identical rows: 1 and 3, and 4 and 6. Row 1 corresponds to state x_1' of Fig. 9.25, and the row 4 to state x_2''. Figure 9.25 shows that these states can only act as initial ones, since there are no paths to them. Therefore, we can further simplify this state diagram by removing these states: thus,

$\varkappa_2'$ can act for $\varkappa_1'$ as an initial state [an identical path (ρ_2, λ_3) leads from $\varkappa_2'$ to $\varkappa_3''$]. Similarly, $\varkappa_2''$ can be replaced by $\varkappa_3''$.

Making similar preliminary simplifications wherever possible, we shall arrive at a diagram with fewer states, that is, at an interconnection matrix of a lower order. In our example, we could have started the algorithmic minimization with a matrix simpler than C''—one with no rows or columns composed exclusively of zeros:

$$C_*'' = \begin{array}{c} \\ \varkappa_1'' \\ \varkappa_2' \\ \varkappa_3' \\ \varkappa_3'' \end{array} \begin{array}{c} \begin{array}{cccc} \varkappa_1'' & \varkappa_2' & \varkappa_3' & \varkappa_3'' \end{array} \\ \left[\begin{array}{cccc} 0 & 0 & (\rho_1, \lambda_3) & 0 \\ 0 & 0 & 0 & (\rho_2, \lambda_3) \\ (\rho_2, \lambda_1) & 0 & 0 & 0 \\ 0 & (\rho_1, \lambda_2) & 0 & 0 \end{array}\right] \end{array}.$$

Then, symmetric decomposition of C_*'' gives the same result as that obtained with C'' as the starting matrix.

Compare now machine N''' (Fig. 9.26) with the initial automaton A (Fig. 9.23): we see that indeed it is the input sequence which governs the equivalence of states of the two machines. Thus, assume the machine of Fig. 9.23 starts up in state $\varkappa_2$, and the input is $\rho_1\rho_2\rho_1\rho_2\rho_1\rho_2\ldots$. Then, to obtain the same output with the machine of Fig. 9.26, the latter must be started from state $\varkappa_2$; if, however, the machine of Fig. 9.23 starts from this state $\varkappa_2$ with an input $\rho_2\rho_1\rho_2\rho_1\rho_2\rho_1\ldots$, then, to obtain the same output from the machine of Fig. 9.26, the latter must be started from state $\varkappa_1$.

10

Transformation of Clock Rates of Sequential Machines

10.1. GENERAL CONSIDERATIONS REGARDING CLOCK RATE TRANSFORMATION. DEFINITION OF REPRESENTATION AND REPRODUCTION

In discussing various practical embodiments of finite automata and sequential machines in Chapter 5, we have singled out a design method whereby an s-machine with a desired clock rate is created on the basis of the equilibrium states of another s-machine, operating at a much faster rate. We shall now return to this problem, and shall analyze it in more general terms. First, however, we shall recall some concepts and definitions of Chapter 5.

Assume we have an s-machine S, to which we feed (at discrete moments 0, 1, ... p) a sequence $\rho^0\ \rho^1\ \ldots\ \rho^p$. We thus obtain the tape of Table 10.1:

Table 10.1

Discrete moment	0	1	2	...	p	...
ρ	ρ^0	ρ^1	ρ^2	...	ρ^p	...
λ	λ^0	λ^1	λ^2	...	λ^p	...

Now we select some sequence of discrete moments, for example, moments 0, 1, 4, and so on, which lie on a continuous scale such that

$$t_0 < t_1 < t_2 < \ldots < t_s . \tag{10.1}$$

We then extract from the tape of Table 10.1 the columns corresponding to this sequence. We thus get Table 10.2:

Table 10.2

t	t_0	t_1	...	t_s	...
ρ	ρ^{t_0}	ρ^{t_1}	...	ρ^{t_s}	...
λ	λ^{t_0}	λ^{t_1}	...	λ^{t_s}	...

We now introduce another clock rate to match our selected sequence of discrete moments, assuming that moment 0 of that sequence occurs at time t_0, moment 1—at time t_1, and so on. We then rewrite Table 10.2 in terms of this new clock rate, and obtain Table 10.3:

Table 10.3

Discrete moment	0	1	...	s	...
ρ	ρ^{t_0}	ρ^{t_1}	...	ρ^{t_s}	...
λ	λ^{t_0}	λ^{t_1}	...	λ^{t_s}	...

The tape of Table 10.3 may be regarded as produced by some new machine G. In fact, if the given tape of machine S (Table 10.1) and, therefore, the tape obtained from it by clock rate transformation (Table 10.3) are both finite, then there must exist an s-machine G producing that last tape (see Section 8.2).

To illustrate this concept, imagine an s-machine whose tape is flash-illuminated at times $t_0, t_1, \ldots, t_s$, corresponding to the sequence of discrete moments of our second clock rate. Machine S will then appear to us to be processing the sequence $\rho^{t_0}, \rho^{t_1}, \ldots, \rho^{t_s}$ into the sequence $\lambda^{t_0}, \lambda^{t_1}, \ldots, \lambda^{t_s}$ in accordance with Table 10.3, whereas in reality it is operating in accordance with Table 10.1, processing the sequence $\rho^0, \rho^1, \ldots, \rho^P$ into the sequence $\lambda^0, \lambda^1, \ldots, \lambda^P$.

Let us now assume that the sequence of times $t_0, t_1, \ldots$, at which the flashes illuminate tape S, is so fortuitously chosen that whatever the input sequence processed by S and whatever its initial state $\varkappa_S^0$, we shall always perceive a sequence of input-output pairs that could be attributed to some s-machine G, which starts up from some state $\varkappa_G^0$ (whereby $\varkappa_G^0$ may vary with each input sequence). If that is the case, we have a *clock rate transformation*—machine S, operating at a rate which, by convention, we shall call *fast*, serves as a basis

for another machine G operating at clock rate which we shall call slow.* We shall also say that the fast machine S *represents* the slow machine G.

These concepts are quite broad, but have a drawback. The point is that the initial state $\varkappa_G^0$ of G is governed not only by the initial state $\varkappa_S^0$ of S, but also by the input sequence $\rho(t)$. This means that at different $\rho(t)$, there will be different $\varkappa_S^0$ for the same $\varkappa_G^0$. Thus to find the appropriate state $\varkappa_G^0$ of G one must not only know beforehand the state $\varkappa_S^0$ of S, but also the input sequence which will be fed into S. This is not unlike the situation encountered in Chapter 9 in connection with the definition of equivalence of s-machines. There the problem was solved by narrowing the concepts of equivalence in such a way that the choice of the initial state did not require an *a priori* knowledge of the input sequence. However, the present authors' attempt to similarly narrow the definitions of representation and transformation of clock rate was unsuccessful. This is because a rigid adherence to a scheme whereby any state $\varkappa_S^0$ of S would always correspond to the same $\varkappa_G^0$ of G, regardless of the input sequence, would have prevented us from investigating several important practical cases of clock rate transformation (we shall return to this question at a later stage and shall then clarify this statement by an example). We shall, therefore, resort to other definitions which are narrower than those above and do not require an *a priori* knowledge of the entire input sequence in order to determine the initial state of the represented machine.

The algorithm for selecting the appropriate time sequence t_0, t_1, t_2, ... synchronizing S and G, will be called the *rule of clock rate transformation*. We shall define it by saying that *the fast machine S represents the slow machine G if for any initial state $\varkappa_S^0$ of S and any input sequence $\rho^0\rho^1\rho^2$ ... there exists at least one initial state $\varkappa_G^0$ of G such that G, starting from this state and processing a sequence $\rho^{t_0}\rho^{t_1}\rho^{t_2}$ ..., will generate a tape coinciding with the image obtained by viewing the tape of S at times t_0, t_1, t_2, ...*.

Given this definition of representation, *state $\varkappa_G^0$ of G is determined by the state $\varkappa_S^0$ of S and the first term of the input sequence to S*.

Note now that the fast machine S, which admits any arbitrary input sequence, usually represents a machine G which can admit inputs only from a restricted set L_G. This means that an image of

*It is convenient, but not necessary, to imagine that the fast machine does indeed operate at a faster clock rate than the slow machine. In general, however, S and G are totally unrelated. Our further discussion shall deal with the general case, in which S and G may operate at any desired clock rates.

the tape of S, obtained by viewing it at $t_0, t_1, t_2, \ldots$, may represent only one of the several possible variants of operation of G. We shall encounter a case of this kind in Section 10.2, where set L_G will consist of sequences containing only one symbol. In general, representation is not a unique relationship, because at any specific clock rate transformation, a given machine S may represent several different machines $G_1, G_2, G_3, \ldots$. This conclusion also holds for the case where there is no restriction on the set of input sequences of G, that is, when $L_G = E$.

By analogy with relative equivalence (see Chapter 9), we can also define relative representation. The definition of relative representation differs from that of representation in general only in that *the fast machine S may not admit arbitrary input sequences but only those belonging to set L_S of sequences allowed in S*. We shall say that in this case *machine S represents machine G in terms of set L_S*.

When $L_S \subseteq E$, then L_G may coincide with L_S, be narrower or broader, intersect with it, etc. In particular, when $L_S = E$, L_G may be restricted, and, conversely, it can happen that $L_S \subset E$ and $L_G = E$.

It is quite obvious that the mode of representation by S of any machine G is closely related to the time sequence $t_0, t_1, t_2, \ldots$ at which the tape of S is viewed. In the general case, this time sequence may be such that S does not represent any sequential machine.

The choice of the (viewing) times $t_0, t_1, t_2, \ldots$ may depend on the input sequence $\rho(t)$, the output sequence $\lambda(t)$, the sequence $\varkappa(t)$ of the states of machine S, as well as the time t.

The "clock," which is a machine that signals the advent of the "slow" discrete (viewing) moments $t_0, t_1, t_2, \ldots$, must allow the input of time t and the symbols $\rho(t)$, $\lambda(t)$, and $\varkappa(t)$ [or some of these symbols], all of which are related to the operation of the fast machine S. The "clock" must be able to perform an algorithm* which processes a given sequence of symbols of the s-machine into the sequence $t_0, t_1, t_2, \ldots$.

We shall assume that the clock itself is a finite automaton with an output converter which operates at the same fast clock rate as the s-machine S. The alphabet of this automaton is obtained by combining all or part of alphabets $\{\rho\}$, $\{\varkappa\}$, and $\{\lambda\}$, depending on the factors determining the sequence $t_0, t_1, t_2, \ldots$. The process of producing a synchronizing signal indicating the advent of a discrete moment such as $t_0, t_1, \ldots$, can then be regarded as a representation of an event at the input of this clock automaton.

*That is, it is a Turing machine (see Chapter 13).

Having defined representation (or relative representation), we are faced with the following problems:

1. Given a machine S, a set L_S, and a clock (that is, an automaton A with a converter Φ), find at least one machine G which can be represented by S in terms of L_S, as well as its set of allowed input sequences L_G.

2. Given a machine S and a machine G, find whether there exists a clock rate transformation such that S will represent G, and if so, determine it (construct automaton A and converter Φ of the clock).

A similar problem also arises with respect to relative representation (here, the set L_G must also be determined).

3. Given a machine S, a set L_S, and the clock rate transformation, construct a minimal machine $G_{\min}$, represented by S in terms of L_S, and find its set of allowed input sequences $L_{G_{\min}}$.

No general solutions to these problems exist as of now, and it is possible that some of them will prove to be algorithmically unsolvable.

In conclusion of our discussion of representation and clock rate transformation, let us note that these concepts could be broadened by permitting the use of converters $\rho^* = \Phi_1(\rho)$ and $\lambda^* = \Phi_2(\lambda)$, in accordance with Fig. 10.1. In this scheme, the input-output pairs occurring at $t_0, t_1, t_2, \ldots$ are not (ρ, λ), but (ρ^*, λ^*). However, we do not need this broader definition for our discussion.

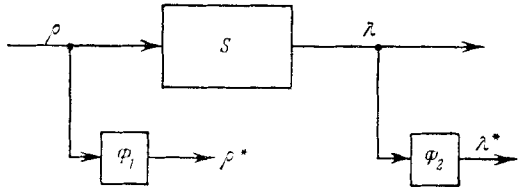


Fig. 10.1.

Assume now that we are given a specific clock rate transformation, a slow machine G and a fast machine S, and the set L_G of G (since G and L_G are given, we can determine the output of G at any input from L_G and at any initial state $\varkappa_G^0$). Now let us define *relative reproduction*. We shall say that S *reproduces* G *in terms of set* L_G *if for any* $\varkappa_G^0$ *of* G *and at any input sequence* $\rho^*(t) = \rho^0\rho^1\rho^2 \ldots$, *from* L_G *there exist at least one initial state* $\varkappa_S^0$ *of* S, *determined by* $\varkappa_G^0$ *and by the first term* ρ^0 *of* $\rho^*(t)$, *as well as at least one sequence* $\tilde{\rho}(t)$ *such that the tape of* S, *which operates under these conditions and is viewed only during the discrete moments of the "slow" time sequence* $t_0, t_1, t_2, \ldots$, *coincides with the tape of* G. When $L_G = E$, relative reproduction and (simple) reproduction become identical.

To avoid confusion, we must stress that representation and reproduction (both relative and nonrelative) are two entirely different and even opposing concepts. Thus representation requires coincidence between each tape of the fast machine S, when viewed at $t_0, t_1, t_2, \ldots,$ and one of the tapes of the slow machine G; on the other hand, reproduction requires that for each tape of G there be a tape of the fast machine S such that when it is viewed at $t_0, t_1, t_2, \ldots,$ it will coincide with the given tape of G. Representation does not imply reproduction, because S may represent G but not reproduce it, and vice versa. Again, reproduction is not unique: for any given specific clock rate transformation there may exist many different fast machines $S_1, S_2, S_3, \ldots,$ each of which will reproduce a given slow machine G.

The set L_S allowed in a fast machine S reproducing a given machine G in terms of L_G,* is also not unique;** and what is more, in many cases L_S may contain symbols which do not appear in L_G.

Indeed, according to the definition of reproduction, for each sequence $\rho^*(t) \in L_G$ and state $\varkappa_G^0$ there will be at least one corresponding sequence $\tilde{\rho}(t)$ allowed as an input to S. However, in the general case, for each $\rho^*(t)$ and $\varkappa_G^0$ there may be not one, but many (possibly even an infinite number) of different sequences $\tilde{\rho}(t)$. These sequences $\tilde{\rho}(t)$, corresponding to all the $\rho^*(t)$ at all possible $\varkappa_G^0$, may form many (possibly even an infinite number) of different sets $L_S^1, L_S^2, L_S^3, \ldots,$ each of the sets L_S^i containing at least one sequence $\tilde{\rho}(t)$ corresponding to any given $\rho^*(t) \in L_G$ and $\varkappa_G^0$.

Each of the sets L_S^i may be considered as a set of inputs allowed in the fast machine S which reproduces G in terms of L_G. Which of these sets is selected depends on additional, practical considerations—sometimes it is convenient to use $L_S = \bigcup_i L_S^i$, and on other occasions set E (which is always usable) is selected as the set of inputs allowed in S.

The concept of reproduction gives rise to the same problems as representation [(1) given a clock and one of the machines, find the other machine; (2) given the two machines, find the clock; and (3) the minimization problem].

In conclusion let us point out that the definition of reproduction entails the same restriction as that of representation: that is, the state $\varkappa_S^0$ of machine S is determined by the state $\varkappa_G^0$ of machine G and only the first term of input $\rho^*(t)$. If $\varkappa_S^0$ were related to $\varkappa_G^0$ and the entire input sequence, we should obtain a broader, but also a

*L_G may coincide with E, that is, the reproduction may be nonrelative.

**This nonuniqueness is not encountered in the case of representation where, given L_S, a specific clock rate transformation, and machine S, the set L_G is uniquely determined.

more inconvenient definition of reproduction [we would have to know, *a priori*, the entire input sequence $\rho^*(t)$].

We shall now clarify representation and reproduction by two simple examples.

10.2. EXAMPLES OF REPRESENTATION AND REPRODUCTION

a. Flip-Flop

Our first example involves the flip-flop of Chapter 5.

Table 10.4

x \ ρ	$\rho_1 = 0$	$\rho_2 = 1$
x_1	x_3	—
x_2	—	x_1
x_3	—	x_4
x_4	x_2	—

Table 10.5

x \ ρ	$\rho_1 = 0$	$\rho_2 = 1$
x_1	x_3	x_4
x_2	x_3	x_1
x_3	x_2	x_4
x_4	x_2	x_1

Consider a P-Pr automaton with a basic Table 10.4. It accepts inputs from set R, containing all sequences in which no two successive symbols are identical. The blanks in the table indicate that the automaton is never in these internal states, so that the corresponding squares may be filled in any desired fashion, for instance, as in Table 10.5. This automaton is diagrammed in Fig. 10.2. We shall observe it only at times $t_0, t_1, t_2 \ldots$, when the state of the input changes from $\rho_1 = 0$ to $\rho_2 = 1$, and shall find which machine G is represented by this automaton in terms of R.

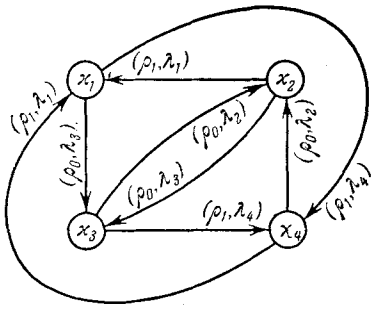


Fig. 10.2.

First of all, let us determine the set L_G of G. Since we observe this automaton only when $\rho = \rho_2 = 1$, L_G will contain only unit sequences. In other words, G will be an autonomous s-machine.

An analysis of operation of the automaton of Fig. 10.2 shows that the s-machine G (or, to be precise, one of the many possible machines G), which the automaton represents

has the state diagram of Fig. 10.3, that is, the slow machine G is a flip-flop (see Section 5.2). The relationship between the various states automaton S and flip-flop G is given in Table 10.6.

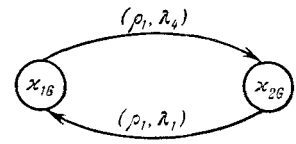


Fig. 10.3.

Now, if the blank squares of Table 10.4 were filled differently, for example, as in Table 10.7, then the relationship between the states of S and G would be represented by Table 10.8.

Table 10.6

S	G	Type of input
x_1	x_{1G}	independent of input sequence
x_2	x_{1G}	if the input sequence begins with $\rho_1 = 0$
x_2	x_{2G}	if the input sequence begins with $\rho_2 = 1$
x_3	x_{2G}	if the input sequence begins with $\rho_1 = 0$
x_3	x_{1G}	if the input sequence begins with $\rho_2 = 1$
x_4	x_{2G}	independent of input sequence

Table 10.7

x \ ρ	$\rho_1 = 0$	$\rho_2 = 1$
x_1	x_3	x_4
x_2	x_2	x_1
x_3	x_3	x_4
x_4	x_2	x_1

Table 10.8

S	G	Type of input
x_1	x_{1G}	independent of input sequence
x_2	x_{2G}	independent of input sequence
x_3	x_{1G}	independent of input sequence
x_4	x_{2G}	independent of input sequence

In this example the clock is a finite automaton which represents the event "ρ_2 occurs after ρ_1." Note that S not only *represents* G with respect to R, but also *reproduces* it in terms of set L_G of unit length input sequences. To achieve reproduction the set L_S of sequences allowed in the automaton can coincide either with E (which contains all sequences of 0 and 1), or with set R, or with any set containing R.

b. Delay Line

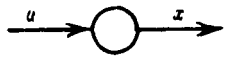
Our second example involves reproduction of a slow s-machine G by a fast machine S built from fast delay elements. Assume we require an s-machine G in which the interval between discrete moments (clock rate) is τ seconds, and where the set L_G coincides with the set E of all allowed input sequences. The input alphabet of G contains r differing characters ρ.

There is no problem in synthesizing a machine G from n-ary delay elements operating in given clock rate τ. However, we have only "fast" n-ary delay elements operating at rate q times faster than τ, that is, at intervals of $\frac{\tau}{q}$, where $q \geqslant 2$. We shall now use these elements to synthesize a fast machine S, and we shall find a clock rate transformation such that S will reproduce G.

The equation for the fast delay element (Fig. 10.4) is

$$x\left(t+\frac{\tau}{q}\right)=u(t), \tag{10.2}$$

while that for the required slow element is



$$x(t+\tau)=u(t). \tag{10.3}$$

Fig. 10.4.

Now, a chain of q fast delay elements (Fig. 10.5) is described by a system of recurrence relations

$$\left.\begin{aligned} x\left(t+\tfrac{\tau}{q}\right)&=x_1(t),\\ x_1\left(t+\tfrac{\tau}{q}\right)&=x_2(t),\\ &\cdots\cdots\cdots\\ x_{q-1}\left(t+\tfrac{\tau}{q}\right)&=u(t). \end{aligned}\right\} \tag{10.4}$$

Eliminating all x_i except x, we get for the entire chain

$$x(t+\tau)=u(t). \tag{10.5}$$

Equation (10.5) coincides with Eq. (10.3) for the slow delay element; therefore, a chain of q fast elements is equivalent to one slow element.

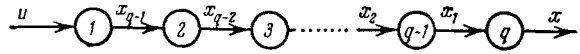


Fig. 10.5.

Bearing this in mind, we construct the s-machine S in the following manner. Assuming for a moment that we have at our disposal the

slow delay elements of Eq. (10.3), we construct from these elements and from instantaneous logical converters a machine G, using one of the methods of Chapter 5. Now, we replace each slow element of G by a chain of q fast delays. The resulting s-machine S will operate at a fast clock rate, that of the fast delay elements. But if S is observed only at $t_0, t_1, t_2, \ldots$, coinciding with moments $0, \tau, 2\tau, 3\tau, \ldots$, then S will reproduce G, since the *cycle* of a fast delay chain coincides with that of one slow element.

The relationship between the states of S and G is independent of the input: for each state $\varkappa_{iG}$ of G there exists such a state of all the fast delays of S at which the state of the initial fast delays of each chain coincides with that of the corresponding slow delays. Calculations show that for each of the r^k states of G there are $r^{k(q-1)}$ states of S reproducing it. The set L_G of S may be either set E, or set R (in which there are no sequences with repeating symbols), or set M which contains all sequences such as

$$\underbrace{\rho_{\alpha_0}\rho_{\alpha_0}\ldots\rho_{\alpha_0}}_{q\text{ times}}\quad\underbrace{\rho_{\alpha_1}\rho_{\alpha_1}\ldots\rho_{\alpha_1}}_{q\text{ times}}\quad\underbrace{\rho_{\alpha_2}\rho_{\alpha_2}\ldots\rho_{\alpha_2}}_{q\text{ times}}\ldots\underbrace{\rho_{\alpha_s}\rho_{\alpha_s}\ldots\rho_{\alpha_s}}_{q\text{ times}}$$
$$\left(\text{possible }\rho_{\alpha_i}=\rho_{\alpha_{i+1}}\right),$$

or another of the many possible sets (all these sets must have the following property: if the symbols in positions $0, q, 2q, 3q, \ldots$ are extracted from each sequence belonging to a given set and arranged into a new set, then this new set must be the set E).

The clock suitable for this case is a ring scaler (which is an autonomous finite automaton) made up of fast delay elements and emitting a signal indicating the occurrence of a "slow" discrete moment every q "fast" moments.

It may be pointed out that in this example (just as in the preceding one) the synthesis was so successful that machine S not only reproduces, but also represents machine G (both in terms of E and in terms of any other set). The relationship between the states of the two machines in the case of representation remains the same as in the case of reproduction.

10.3 REPRODUCTION OF A SLOW MACHINE ON A FAST ONE IN THE CASE WHEN THE CYCLE OF THE SLOW MACHINE IS GOVERNED BY THE CHANGE OF INPUT STATE

This problem mentioned was already discussed in Chapter 5, where we arrived at a solution. We shall produce here another solution which is suitable for any machine.

Suppose we are given some slow s-machine G whose cycle (that is, clock rate) is governed by change of input. Let this G be given as an interconnection matrix or a state diagram. The set L_G of G contains all the possible sequences except those with two identical symbols in a row. We need a fast s-machine S which reproduces G in terms of L_G, and whose clock rate is related to that of G in the following manner: G operates at instants $t_0, t_1, t_2, \ldots, t_s$ which occur when S reaches equilibrium after any change of input. Assume that the maximum number of fast cycles necessary for S to go from one equilibrium state to another upon a change of input is m. We shall then assume that, at reproduction, the set L_G of S contains all the sequences such as

$$\underbrace{p_{\alpha_0}p_{\alpha_0}\cdots p_{\alpha_0}}_{q_0 \text{ times}}\ \underbrace{p_{\alpha_1}p_{\alpha_1}\cdots p_{\alpha_1}\cdots}_{q_1 \text{ times}}\ \underbrace{p_{\alpha_i}p_{\alpha_i}\cdots p_{\alpha_i}}_{q_i \text{ times}}\ \left(p_{\alpha_i} \neq p_{\alpha_{i+1}}\right), \tag{10.6}$$

where $q_i \geqslant m$ for all $i = 1, 2, 3, \ldots$. This means that an input to S cannot change until the machine is in equilibrium.

Assuming that $q_i \geqslant q^*$, the set of sequences such as (10.6), will be denoted by T_{q^*}. The sets T_{q^*} satisfy the relationship

$$T_1 \supset T_2 \supset T_3 \supset \ldots \supset T_{q^*} \supset \ldots, \tag{10.7}$$

whereby $T_1 = E$. Thus, provided $q^* \geqslant m$, any set T_q^* can serve as the set L_S of S.

If the condition of replacement of G by S specifies that the two machines must operate synchronously, then condition $q^* \geqslant m$ means that there are at least m cycles of S between two successive cycles of G.

We shall construct machine S by transforming the state diagram of the given machine G. Assume that state $\varkappa_i$ of this diagram has the form of Fig. 10.6. We shall replace $\varkappa_i$ with as many states as there are different paths terminating in that state (a loop path is considered to be both terminating and originating in state $\varkappa_i$). This gives the four states $\varkappa_{i1}, \varkappa_{i2}, \varkappa_{i3}, \varkappa_{i4}$ (surrounded by a dotted line) of Fig. 10.7, where each of these new states also carries a loop path labeled in the same way as the path terminating in that state.

From each of the new states we draw the same paths as those which originated in state $\varkappa_i$ of Fig. 10.6; however, we need not draw those paths carrying the same p symbol as the loop at the state from which that path would originate (see Fig. 10.7).

We do the same thing for all states of G, and obtain the state diagram of a machine S which reproduces G in terms of L_G, the relationships between the states of G and S being as follows (Figs. 10.6

and 10.7): at $\rho = \rho_s$, state x_{j1} of G corresponds to state x_{i1} of S, where x_{i1} is the equilibrium state of S at input ρ_s; obviously, at input ρ_s the same state x_{i1} of S also corresponds to state x_i of G; similarly, at $\rho = \rho_p$ the state x_{j2} of G corresponds to state x_{i2} of S, x_{i2} being the state of equilibrium of S at $\rho = \rho_p$, and so on. The general correspondence is established in a similar manner. Now it is readily seen from the state diagram that S goes from one state of equilibrium to another at any change of the input and that this transition is accomplished in one "fast" cycle. That is, $m = 1$. In addition, the diagram shows that machine S has no unstable states at any input, because each state has a loop path and therefore is a state of equilibrium for some specific input.

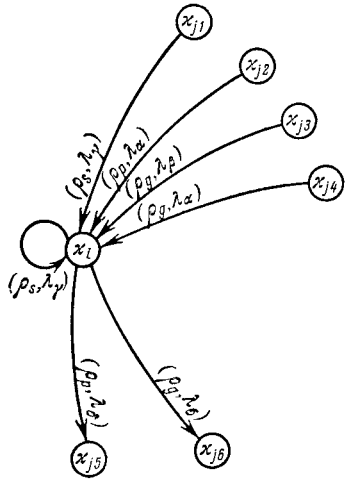


Fig. 10.6.

An input sequence of the reproducing machine S, corresponding to the input sequence $\rho_{\alpha_0}\rho_{\alpha_1}\rho_{\alpha_2} \cdots \rho_{\alpha_i} \cdots \left(\rho_{\alpha_i} \neq \rho_{\alpha_{i+1}}\right)$ of the slow machine G, has the form of Eq. (10.6), where all the $q_i \geqslant 1$. Using $q = 1$, we find that one of the (corresponding) input sequences of S is $\rho_{\alpha_0}\rho_{\alpha_1}\rho_{\alpha_2} \ldots$, that is, it coincides with the input sequence to G.

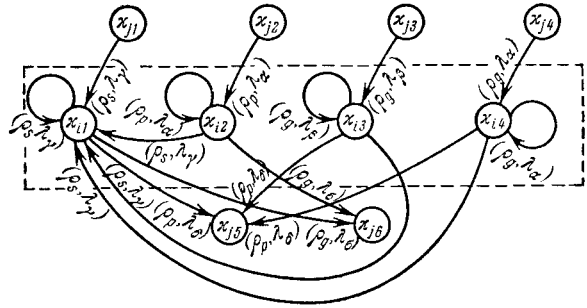


Fig. 10.7.

The instants t_0, t_1, t_2, ..., at which the fast machine S is "viewed" (that is, at which information is extracted from the tape of S) occur one "fast" cycle after the change of the state of input of S. If the input sequence of S is made to coincide with that of G, then information is extracted from S in all fast cycles.

Thus, our transformation of the state diagram of G solves the problem of synthesis of a fast s-machine S which reproduces, in

terms of L_G, a given s-machine G responding to the change of the input state. Here we used the state diagram only for better visualization of the problem. In practice, it may be more convenient to transform the interconnection matrix. This is done as follows:

Assume the interconnection matrix C^G of G is given, and let it conform to Fig. 10.6, so that its ith row and column are

$$C^G = \begin{array}{c} \\ \\ j1 \\ j2 \\ \\ \\ i \\ \\ \\ \\ j3 \\ \\ \\ j4 \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccc} \cdots & i & \cdots & j5 & \cdots & j6 & \cdots \end{array} \\ \left[\begin{array}{ccccccc} & \cdots & & & & & \\ & (\rho_s, \lambda_\gamma) & & & & & \\ & (\rho_p, \lambda_\alpha) & & & & & \\ & \cdots & & & & & \\ \cdots & (\rho_s, \lambda_\gamma) & \cdots & (\rho_p, \lambda_\delta) & \cdots & (\rho_g, \lambda_\sigma) & \cdots \\ & \cdots & & & & & \\ & (\rho_g, \lambda_\beta) & & & & & \\ & \cdots & & & & & \\ & (\rho_g, \lambda_\alpha) & & & & & \\ & \cdots & & & & & \end{array}\right] \end{array}.$$

We shall now replace this row and column with as many rows and columns as there are different pairs in the ith column of C^G. Since there are four such pairs $[(\rho_s, \lambda_\gamma), (\rho_p, \lambda_\alpha), (\rho_g, \lambda_\beta)$, and $(\rho_g, \lambda_\alpha)]$, we introduce the rows and columns $i1$, $i2$, $i3$ and $i4$:

$$C = \begin{array}{c} \\ j1 \\ \\ \\ j2 \\ \\ \\ i1 \\ i2 \\ i3 \\ i4 \\ \\ \\ j3 \\ \\ \\ j4 \\ \\ \end{array} \begin{array}{c} \begin{array}{cccccccc} & i1 & i2 & i3 & i4 & \cdots & j5 & \cdots & j6 \end{array} \\ \left[\begin{array}{cccccccc} \cdots & (\rho_s, \lambda_\gamma) & & & & & & & \\ & & & & & & & & \\ & & (\rho_p, \lambda_\alpha) & & & & & & \\ & & & & & & & & \\ \cdots & (\rho_s, \lambda_\gamma) & & & & & (\rho_p, \lambda_\delta) & (\rho_g, \lambda_\sigma) & \\ \cdots & (\rho_s, \lambda_\gamma) & (\rho_p, \lambda_\alpha) & & & & & (\rho_g, \lambda_\sigma) & \\ \cdots & (\rho_s, \lambda_\gamma) & & (\rho_g, \lambda_\beta) & & & (\rho_p, \lambda_\delta) & & \\ \cdots & (\rho_s, \lambda_\gamma) & & & (\rho_g, \lambda_\alpha) & & (\rho_p, \lambda_\delta) & & \\ & & & & & & & & \\ & & & & & & & & \\ & & & (\rho_g, \lambda_\beta) & & & & & \\ & & & & & & & & \\ & & & & & & & & \\ & & & & (\rho_g, \lambda_\alpha) & & & & \\ & & & & & & & & \\ & & & & & & & & \end{array}\right] \end{array}$$

For diagonal elements of the submatrix comprising these four rows and columns we take (in any desired order) all the different pairs of the ith column of C^G. Furthermore, the intersection of row $j1$ and column i of C^G contains the pair (ρ_s, λ_γ); we retain it in row $j1$ of the new matrix C, but place it in the column where (ρ_s, λ_γ) is already present, that is, column $i1$. We do the same with the pairs of rows $j2$, $j3$ and $j4$ of the ith column of C^G. Now the columns $i1$—$i4$ are complete, and each contain only identical pairs. We then fill in rows $i1$—$i4$ of C as follows: all the pairs of the ith row of C^G [except for pair (ρ_s, λ_γ) which is present on the diagonal of C^G] are transposed into all four rows ($i1$—$i4$), retaining these pairs in the same columns as in C^G. However, if the ρ symbol of the pair being transposed into a given row coincides with the ρ symbol of a pair already present in that row, then there is no need for this transposition—the space is left blank. The pair (ρ_s, λ_γ) is transposed into all rows $i1$—$i4$, being placed in that column in which it already appears as a result of filling in the disposal elements. To be specific, the ith row of our example of C^G contains pairs (ρ_s, λ_γ), (ρ_p, λ_δ), (ρ_g, λ_σ). Row $i1$ of C already contains pair (ρ_s, λ_γ); we transpose into it pair (ρ_p, λ_δ) and place it in column $j5$, and the pair (ρ_g, λ_σ) in column $j6$. We add to the row $i2$ the pair (ρ_s, λ_γ) in column $i1$ and the pair (ρ_g, λ_σ) in the column $j6$ [pair (ρ_p, λ_δ) is omitted from column $j5$ since row $i2$ already contains pair (ρ_p, λ_α)]. Rows $i3$ and $i4$ are filled in the same manner.

The above procedure must be repeated for all the ith rows and columns of C^G. As a result, we obtain a matrix C which actually is the interconnection matrix C^S of the fast machine S. Note one property of C^S: *all columns of this matrix contain (only) identical pairs*.

It is obvious that the transformation of the interconnection matrix C^G into C^S is a procedure identical to that employed in the previously described transformation of state diagrams.

Let us conclude this section with two notes.

Note 1. In this section, just as in Section 10.2, we devised the machine S so "successfully" that it not only reproduces but also represents machine G. In representation, the set of allowed input sequences may be any set, including set E, since the inputs to S can change at a rate coinciding with the clock rate of the fast machine S. This is because m of S is 1. In representation, the correspondences between the states of G and S are the same as in reproduction.

Note 2. The above technique is only one of the available methods for synthesizing a fast s-machine S, reproducing a given s-machine G in terms of L_G. Other techniques are also possible, since S is not the only machine reproducing G with respect to L_G. For this reason there arises the problem of minimization of S, that is, the problem

of synthesizing the machine S in such a way that it will contain a minimal number of states.

10.4. MINIMIZATION OF THE s-MACHINE OF SECTION 10.3

We shall minimize the machine of Section 10.3, that is, synthesize a machine $S_{\min}$ reproducing the given machine G in terms of L_G, but having the least possible number of internal states. The required machine $S_{\min}$ will have to satisfy two conditions.

Condition 1. Each state of $S_{\min}$ must be an equilibrium state for at least one input.

Condition 2. Regardless of what changes are made at the input, $S_{\min}$ must reach a new equilibrium in one fast cycle (that is, $m_{S_{\min}} = 1$).

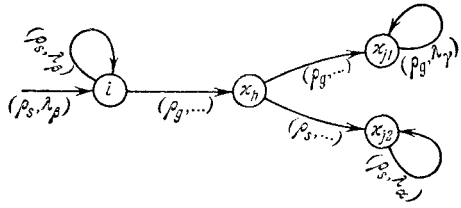


Fig. 10.8.

We shall now prove that these conditions do not restrict the generality of our minimization. Assume that machine $S_{\min}$ does not satisfy condition 1. This would mean that $S_{\min}$ has at least one state x_h which is not a state of equilibrium, and is represented in the state diagram by a circle h not associated with a loop path (Fig. 10.8). If this is so, we can drop this state x_h from the diagram, replacing the path labeled $(\rho_g,\ldots.)$* from x_i to x_{j1} (and passing through x_h) by a direct path $(\rho_g,\ldots)$ from x_i to x_{j1}. The path $(\rho_s,\ldots)$ from x_h to x_{j2}, may be dropped because no path with the same label terminates in x_h. We thus obtain Fig. 10.9, from which all the nonequilibrium states have been removed.

This transformation modifies the operation $S_{\min}$ only during the interval between two equilibria of $S_{\min}$, when we do not care what the machine does anyway. The order in which the equilibrium states change remains unaltered and consequently the modified machine will reproduce the given machine G in the same way as before. However, the very fact that we are able to reduce the number of states in $S_{\min}$ by this transformation contradicts our statement that $S_{\min}$ is a minimal machine. Therefore a minimal machine $S_{\min}$ must satisfy condition 1.

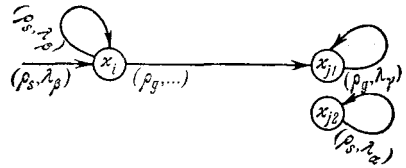


Fig. 10.9.

*Dots in the label indicate that the output symbol can be arbitrary.

In general, for any given machine G there may exist several different minimal machines S_{min}, each reproducing G in terms of L_G. However, all these machines must have the same number of states k_{min}. Our minization problem will be solved when we shall find at least one of these machines.

Let us now turn to condition 2. We shall prove the following statement: if there exists a machine S_{min}, with k_{min} internal states, reproducing G in terms L_G and not satisfying condition 2, then there must exist another machine $\tilde{S}_{min}$ with the same number of internal states k_{min}, which also reproduces G in terms of L_G but which satisfies condition 2. This will show that condition 2 does not restrict the generality of the solution. To prove this statement we shall show that the state diagram of $\tilde{S}_{min}$ can be obtained from that of S_{min} by a transformation which does not alter the number of states.

Let S_{min} go from state $\varkappa_i$, which is an equilibrium for $\rho = \rho_p$, to state $\varkappa_j$ which is an equilibrium for $\rho = \rho_g$. Let this be accomplished in m fast cycles. Then S_{min} will go through $(m-1)$ intermediate states $\varkappa_{i1}, \varkappa_{i2}, \ldots, \varkappa_{i(m-1)}$ (because none of these is an equilibrium state, they cannot contain closed loops). For example, Fig. 10.10 illustrates what happens in a section of some machine at $m = 3$.

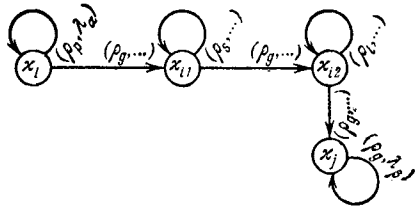


Fig. 10.10.

If the circles i, $i1$, $i2$, ..., $i(m-1)$ are directly connected to circle j by paths labeled (ρ_g, λ_β), then we obtain the state diagram of Fig. 10.11. Now ρ_g shifts the machine from state $\varkappa_i$ and from states $\varkappa_{i1}, \varkappa_{i2}, \ldots, \varkappa_{i(m-1)}$ to the state $\varkappa_j$ in one fast cycle. If we transform

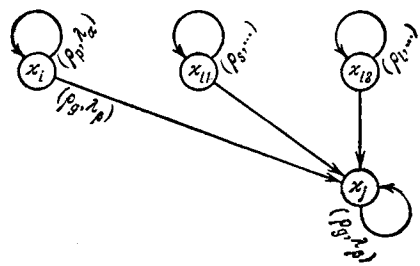


Fig. 10.11.

the entire state diagram of S_{min} in the same manner, then the state diagram of the resulting machine $\tilde{S}_{min}$ will have the same number of states. The operation of S_{min} will differ from that of $\tilde{S}_{min}$ only during the intervals between equilibria, when we do not care what these machines do anyway. However, an allowable input will change the states of equilibrium in $\tilde{S}_{min}$ input sequences in the same way as in S_{min}. This proves our statement.

Having proved that conditions 1 and 2 are not restrictive, let us look for our minimal machine among the machines S_1, S_2, S_3, ..., which reproduce the given machine G in terms of L_G and which satisfy these conditions.

As we have already pointed out in the preceding section, reproduction requires that for any input sequence

$$\rho_{\alpha_0}\rho_{\alpha_1}\rho_{\alpha_2}\cdots\rho_{\alpha_s}\cdots \qquad (\rho_{\alpha_i}\neq\rho_{\alpha_{i+1}}) \tag{10.8}$$

of machine G there be an input sequence

$$\underbrace{\rho_{\alpha_0}\rho_{\alpha_0}\cdots\rho_{\alpha_0}}_{q_0\text{ times}}\ \underbrace{\rho_{\alpha_1}\rho_{\alpha_1}\cdots\rho_{\alpha_1}}_{q_1\text{ times}}\cdots\underbrace{\rho_{\alpha_1}\rho_{\alpha_s}\cdots\rho_{\alpha_s}}_{q_s\text{ times}}\ (\rho_{\alpha_i}\neq\rho_{\alpha_{i+1}})$$

of machine S, where $q_i\geqslant m$ $(i=1, 2, \ldots)$. Since $m=1$ for any S_i, the set L_S of input sequences allowed in S_i can be any set T_{q^*} $(q^*=1, 2, 3, \ldots)$ of sequences such as (10.6), assuming $q_i\geqslant q^*$ $(i=1, 2, 3, \ldots)$; in particular, L_S can be the set $E=T_1$.

Assume we want S_i to reproduce G, and let the set $T_{\bar{q}^*}$ be used as the set L_S. We shall then prove the following statement (A): if state $\varkappa_{\alpha_i}$ of machine S_α and state $\varkappa_{\beta_j}$ of machine S_β are equivalent in terms of set $T_{\bar{q}^*}$, then they are also equivalent with respect to all the sets $T_1, T_2, T_3, \ldots$; that is, they are ***simply equivalent***, since $T_1=E$ (S_α and S_β may also refer to the same machine). If $q>\bar{q}^*$, our statement is obviously true, since in this case $T_q\subset T_{\bar{q}^*}$ [see Section 10.3, Eq. (10.7)]. If $q<\bar{q}^*$, the truth of our statement follows from the fact that with any change of input, all machines S_i will go to an equilibrium in one discrete instant, after which repetition of an input symbol will not change the state of the machine, regardless of the number of times this symbol is fed to the machine.

Every machine S_i reproduces G in terms of set L_G, that is, in terms of the formula of Eq. (10.8). This means that given any initial state $\varkappa_G^0$ and any allowed input to G, there must exist an initial state $\varkappa_{S_i}^0$ of S_i at which that machine, accepting a corresponding input from set $T_{\bar{q}^*}$ [in the form of Eq. (10.6)] and observed only upon attainment of equilibrium after a change of input, generates the same output as G. Without sacrificing generality, we may assume that $\varkappa_{S_i}^0$ will be a state of equilibrium for S_i at $\rho=\rho_{\alpha_0}$; were this not so, then S_i would go in one "fast" cycle from $\varkappa_{S_i}^0$ to $\varkappa_{S_i}^1$ which would have to be a state of equilibrium at $\rho=\rho_{\alpha_0}$. In that case, $\varkappa_{S_i}^1$ would be the state corresponding to $\varkappa_G^0$ of G.

Now consider any two machines S_i and S_j from the set $S_1, S_2, S_3, \ldots$. Suppose that at $\rho=\rho_{\alpha_0}$, state $\varkappa_G^0$ of G corresponds to state $\varkappa_{S_i}^0$ of S_i and to the state $\varkappa_{S_j}^0$ of S_j. Then, since states $\varkappa_{S_i}^0$ and $\varkappa_{S_j}^0$ correspond to the same state of G for the same ρ_{α_0}, the outputs of S_i and S_j, which start from $\varkappa_{S_i}^0$ and $\varkappa_{S_j}^0$, respectively, will coincide for any input from $T_{\bar{q}^*}$ which begins with ρ_{α_0}, this coincidence occurring one "fast" cycle after a change of input. But if there is no change of input, then the machine is also in equilibrium at all other times, and the outputs at these times will also coincide. Thus, the states

$\varkappa^0_{S_i}$ and $\varkappa^0_{S_j}$ are equivalent at all those input sequences from $T_{\bar{q}^*}$ which begin with ρ_{α_0}. But these states will also be equivalent in terms of set $T_{\bar{q}^*}$, since they are equilibrium states at $\rho = \rho_{\alpha_0}$. Indeed, if the input sequence were to begin with some $\rho_{\beta_0} \neq \rho_{\alpha_0}$, then it would be possible to write the sequence

$$\underbrace{\rho_{\alpha_0}\rho_{\alpha_0} \cdots \rho_{\alpha_0}}_{\bar{q}^* \text{ times}}\rho_{\beta_0}, \tag{10.9}$$

which does begin with ρ_{α_0} and does belong to $T_{\bar{q}^*}$. With respect to this sequence, the states $\varkappa^0_{S_i}$ and $\varkappa^0_{S_j}$ are equivalent. At the end of $\bar{q}^*$ cycles, the machines starting from these states, are again in these states; thus the initial conditions are not changed, and we may take the $(\bar{q}^* + 1)$-th fast cycle as the reference time. If we do that, then the sequence (10.9), taken as of the $(\bar{q}^* + 1)$-th fast cycle, begins with ρ_β.

Thus we have shown that all the states of S_i $(i = 1, 2, 3, \ldots)$ corresponding to the same state $\varkappa^0_G$ of G, are equivalent with respect to the chosen set $T_{\bar{q}^*}$ at $\rho = \rho_{\alpha_0}$. It then follows from statement (A) proved above that they are also equivalent with respect to $T_1 = E$, that is, they are simply equivalent.

Let us now assume that set $S_1, S_2, S_3, \ldots$ yields a fast machine $\bar{S}$ which not only reproduces G in terms of L_G, but also represents it in terms of E. This means that for each state of $\bar{S}$ and for any $\rho = \rho_{\alpha_0}$ we can find a state of G, such that there is representation.

Let us take any state $\tilde{\varkappa}_{\bar{S}}$ of $\bar{S}$, and let $\tilde{\varkappa}_{\bar{S}}$ be an equilibrium state for $\rho = \rho_{\alpha_0}$. Then, under representation, the corresponding state of G is $\tilde{\varkappa}_G$. Observing the operation of $\bar{S}$ and G of various inputs to $\bar{S}$ beginning with ρ_{α_0}, we come to the conclusion that if $\tilde{\varkappa}_G$ corresponds, at $\rho = \rho_{\alpha_0}$, to $\tilde{\varkappa}_{\bar{S}}$ under representation, then $\tilde{\varkappa}_{\bar{S}}$ corresponds, at the same ρ_{α_0}, to $\varkappa_G$ under reproduction (that is, machine $\bar{S}$ represents and reproduces machine G). Since $\tilde{\varkappa}_{\bar{S}}$ may be any state of $\bar{S}$, it follows that for each state $\tilde{\varkappa}_{\bar{S}}$ of $\bar{S}$ there exist suitable $\tilde{\varkappa}_G$ and ρ_{α_0} of G such that $\tilde{\varkappa}_{\bar{S}}$ reproduces $\varkappa_G$ at $\rho = \rho_{\alpha_0}$. But from this it follows directly that for any state $\tilde{\varkappa}_{\bar{S}}$ of $\bar{S}$ there exists a state $\tilde{\varkappa}_{S_i}$ equivalent to it in any machine S_i. This means that all machines S_i may be mapped onto machine $\bar{S}$ (some of these S_i may, of course, also be equivalent to $\bar{S}$). Therefore, all we have to do is to minimize machine $\bar{S}$, that is construct a minimal sequential machine $S_{\min}$ equivalent to $\bar{S}$. And this can be done by means of the Aufenkamp-Hohn algorithm (see Section 9.6).* The machine $S_{\min}$ so obtained will be the minimal s-machine reproducing G in terms of L_G.

*For machine S_i, the decomposition of all the states into groups equivalent in terms of set $T_{\bar{q}^*}$ coincides with groupings equivalent in terms of E.

As already pointed out in Section 10.2, the machine S, derived by transforming the interconnection matrix of G, both reproduces G in terms of L_G and represents it in terms of E. This machine also satisfies conditions 1 and 2 of the present section. Consequently, *to obtain* $S_{\min}$ (to be precise, one of the possible minimal machines) *it is sufficient to minimize* S *by symmetrical decomposition of its interconnection matrix*. The result of the minimization does not depend on which of the sets $T_{\bar{q}^*}$ is used as the set L_S of input sequences allowed in S under reproduction. In this case, restricting the number of sets of input sequences does not further reduce the number of states of the reproducing machine.

We shall now construct a minimal s-machine $S_{\min}$ reproducing a given machine G in terms of L_G.

Example. Let the interconnection matrix C^G of a given "slow" machine G operating in alphabets $\{\rho\} = \{1, 2, 3\}$, $\{\varkappa\} = \{1, 2, 3\}$ and $\{\lambda\} = \{1, 2\}$ be

$$C^G = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} (1,1) & (2,1) & (3,1) \\ (2,2)\vee(1,2) & 0 & (3,1) \\ (2,2)\vee(1,1) & (3,2) & 0 \end{bmatrix} \end{array}.$$

The state diagram of G is shown in Fig. 10.12.

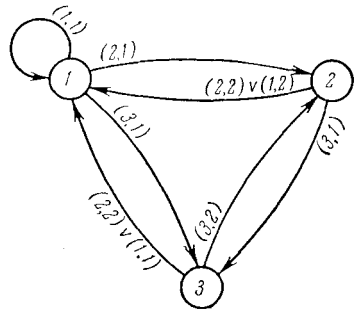


Fig. 10.12.

Transforming C^G as in Section 10.3, we obtain the matrix C^S of the "fast" machine S reproducing G:

$$C^S = \begin{array}{c} \\ 1^1 \\ 1^2 \\ 1^3 \end{array} \begin{array}{c} \begin{array}{cccccc} 1^1 & 1^2 & 1^3 & 2^1 & 2^2 & 3 \end{array} \\ \begin{bmatrix} (1,1) & 0 & 0 & (2,1) & 0 & (3,1) \\ (1,1) & (2,2) & 0 & 0 & 0 & (3,1) \\ 0 & 0 & (1,2) & (2,1) & 0 & (3,1) \end{bmatrix} \end{array}$$

$$\begin{array}{c|cccccc} 2^1 & 0 & 0 & (1,2) & (2,1) & 0 & (3,1) \\ 2^2 & 0 & (2,2) & (1,2) & 0 & (3,2) & 0 \\ 3 & (1,1) & (2,2) & 0 & 0 & 0 & (3,1) \end{array}$$

The state diagram of S is shown in Fig. 10.13. Now let us minimize S. In C^S, rows 2 and 6, as well as 3 and 4 form 1-matrices.

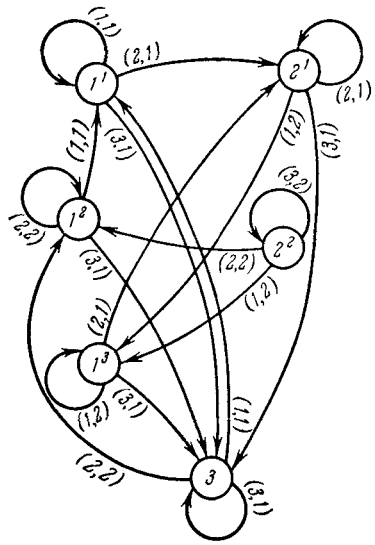


Fig. 10.13.

Let us rewrite C^S so that these rows appear one after another, and then carry out the symmetrical grouping:

$$C^S = \begin{array}{c|cc|cc|cc} & 1^3 & 2^1 & 1^2 & 3 & 1^1 & 2^2 \\ \hline 1^3 & (1,2) & (2,1) & 0 & (3,1) & 0 & 0 \\ 2^1 & (1,2) & (2,1) & 0 & (3,1) & 0 & 0 \\ \hline 1^2 & 0 & 0 & (2,2) & (3,1) & (1,1) & 0 \\ 3 & 0 & 0 & (2,2) & (3,1) & (1,1) & 0 \\ \hline 1^1 & 0 & (2,1) & 0 & (3,1) & (1,1) & 0 \\ \hline 2^2 & (1,2) & 0 & (2,2) & 0 & 0 & (3,2) \end{array}.$$

After all the intermediate steps, we get the interconnection matrix $C^S_{\min}$ of the minimal machine $S_{\min}$ with four states $\varkappa_1, \varkappa_2, \varkappa_3, \varkappa_4$:

$$C^S_{\min} = \begin{array}{c} \\ \varkappa_1 \\ \varkappa_2 \\ \varkappa_3 \\ \varkappa_4 \end{array} \begin{array}{c} \begin{array}{cccc} \varkappa_1 & \varkappa_2 & \varkappa_3 & \varkappa_4 \end{array} \\ \begin{bmatrix} (1,2)\vee(2,1) & (3,1) & 0 & 0 \\ 0 & (2,2)\vee(3,1) & (1,1) & 0 \\ (2,1) & (3,1) & (1,1) & 0 \\ (1,2) & (2,2) & 0 & (3,2) \end{bmatrix} \end{array}.$$

The state diagram of $S_{\min}$ is given in Fig. 10.14.

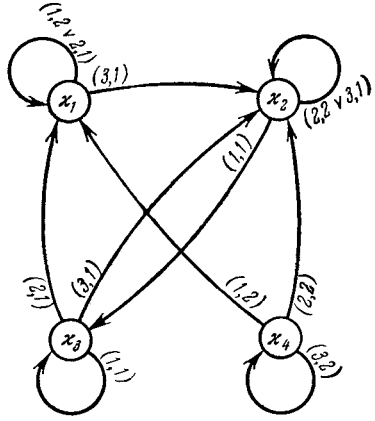


Fig. 10.14.

By virtue of the previously noted fact that each column of C^S can contain only identical pairs, matrix $C^S_{\min}$ has the following property: no column of $C^S_{\min}$ can contain two pairs with identical first and different second subscripts. This means that the state diagram of $S_{\min}$ may be treated, at will, either as the diagram of an s-machine of the P - P type, or as the diagram of an s-machine of the P - Pr type (see the note in Section 3.4). Assume, for example, that the diagram is that of a P - Pr machine. Now, let us show the construction of a relay circuit realizing this machine. To start with, the state diagram yields the tables of the automaton and of the converter of $S_{\min}$ (Tables 10.9 and 10.10).

Table 10.9 can be regarded as a Huffman flow table; all we have to do is to draw squares around the equilibrium states $\tilde{\varkappa}_{\tilde{S}}$; that is, those states whose subscripts are the same as the ordinal numbers of the matrix rows. After this Table 10.9 assumes the form of Table 10.11. From Tables 10.10 and 10.11 we can design a relay circuit realizing $S_{\min}$ by means of the method described in Section 5.4. To do this, we assign binary numbers to the symbols $\varkappa$, ρ and λ, as shown in Tables 10.12 - 10.14. Then Tables 10.9 and 10.10 can be expressed in the form of Tables 10.15 and 10.16, from which we derive the combined Table 10.17.

Table 10.9

α \ ρ	ρ_1	ρ_2	ρ_3
α_1	α_1	α_1	α_2
α_2	α_3	α_2	α_2
α_3	α_3	α_1	α_2
α_4	α_1	α_2	α_4

Table 10.10

α \ ρ	ρ_1	ρ_2	ρ_3
α_1	λ_2	λ_1	λ_1
α_2	λ_1	λ_2	λ_1
α_3	λ_1	λ_1	λ_1
α_4	λ_2	λ_2	λ_2

Table 10.17 defines three logical functions Y_1, Y_2 and Z of four independent variables x_1, x_2, y_1, and y_2. We shall now assume x_1 and x_2 represent the states of the input contacts, y_1 and y_2 —the states of contacts of the secondary relays, and Y_1 and Y_2—the states of the

Table 10.11

α \ ρ	ρ_1	ρ_2	ρ_3
α_1	$\boxed{\alpha_1}$	$\boxed{\alpha_1}$	α_2
α_2	α_3	$\boxed{\alpha_2}$	$\boxed{\alpha_2}$
α_3	$\boxed{\alpha_3}$	α_1	α_2
α_4	α_1	α_2	$\boxed{\alpha_4}$

Table 10.12

α \ y	y_2	y_1
α_1	0	0
α_2	0	1
α_3	1	0
α_4	1	1

Table 10.13

ρ \ x	x_2	x_1
ρ_1	0	0
ρ_2	0	1
ρ_3	1	0

Table 10.14

λ \ Z	Z
λ_1	0
λ_2	1

coils (energized or deenergized) of these secondary relays; the state of the coil of the output relay will be Z. Now, any network made up of contacts x_1, x_2, y_1, and y_2, as well as coils Y_1, Y_2, and Z and realizing the table of logical functions of Table 10.17 will also realize machine $S_{\min}$. A network of this type may be constructed by any method of Chapter 2.

Table 10.15

y \ x	00	01	10
00	00	00	01
01	10	01	01
10	10	00	01
11	00	01	11

Table 10.16

y \ x	00	01	10
00	1	0	0
01	0	1	0
10	0	0	0
11	1	1	1

Now let us compare our minimization method with that of Huffman (Section 5.4). The main difference between the two methods is that

Table 10.17

y_1	0	1	0	1	0	1	0	1	0	1	0	1
y_2	0	0	1	1	0	0	1	1	0	0	1	1
x_1	0	0	0	0	1	1	1	1	0	0	0	0
x_2	0	0	0	0	0	0	0	0	1	1	1	1
Y_1	0	0	0	0	0	1	0	1	1	1	1	1
Y_2	0	1	1	0	0	0	0	0	0	0	0	1
Z	1	0	0	1	0	1	0	1	0	0	0	1

there are no restrictions on the applications of our method, while that of Huffman (as already pointed out in Chapter 5) may only be used to construct those s-machines in which the next state of the automaton of this machine is uniquely determined by the present states of the input and the output of the machine. Where both methods are applicable, they yield identical results, even after minimization.

In concluding this section, let us point out that the algorithm for deriving additional states (Section 10.3) is also applicable when the given slow machine G is subject to Aufenkamp-type constraints. The machine S (which reproduces G in terms of L_G) constructed by means of this algorithm will also be subject to the same constraints. Therefore it should be minimized by the technique described at the end of Section 9.8 (or some other method for full minimization of machines subject to Aufenkamp-type constraints). The fact that this

minimization of the fast machine S gives a minimal machine reproducing G may be proved by the same reasoning as that given in the present section, assuming no constraints (the only additional requirement is finding the set of input sequences allowed for each of the states).

11

Determination of the Properties of Sequential Machines from Their Response to Finite Input Sequences

11.1. DEFINITIONS AND STATEMENT OF PROBLEM

We shall now consider finite automata and s-machines as objects on which one can experiment but about whose internal structure one possesses only limited information. It is also assumed that the experiments can only consist of observing the outputs generated by these machines in response to finite inputs. Our problem is to determine the specific structure of a given finite automaton or s-machine, its present state and, if possible, its state diagram.

We shall say that by feeding a sequence of (finite) length l to the s-machine we are performing an *experiment of length* l. The input of sequence $\rho(t) = \rho^0\rho^1 \ldots \rho^p$ produces a synchronous output of the sequence $\lambda(t) = \lambda^0\lambda^1 \ldots \lambda^p$, which we shall call the *response* of the s-machine to the input of $\rho(t)$. In this chapter, we shall call the input and the corresponding output, that is, the tape of the s-machine, the *result of the experiment*.

One can perform a variety of experiments. Thus when only one s-machine of a given type is available, and the input is a predetermined sequence, we have a *simple nonbranching experiment*. If, however, each consecutive input symbol selected by the experimenter depends on the preceding output symbols, then the experiment is said to be *simply* branching (or just branching). When several identical s-machines are available and all are in the same initial state, one can perform a multiple experiment, whereby different inputs are fed to each machine. A variant of the multiple experiment is one in which there is a single s-machine equipped with a *reset* button, that is, a device returning the machine to its initial state upon completion of an experiment.

The problem of determining the specific structure of a given s-machine from the results of a finite experiment can be defined only after all the *a priori* known facts about this machine have been

exactly stated. As will be shown below, any new data about this s-machine which can be produced by the experiment depend on this *a priori* known information.

At the outset we can make the following intuitively obvious statement: if we do not know anything about a given s-machine, then there is no finite experiment which will tell us even as much as the number of its states. Obviously, to study a given machine we must know beforehand the nature and the number r of the input symbols ρ.

Let S be an s-machine with k internal states $\varkappa_1, \varkappa_2, \ldots, \varkappa_k$ (where k is unknown!) which we subject to a finite experiment of length l. Then it is always possible to devise another s-machine S^* which has

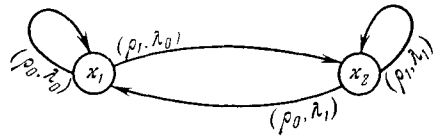


Fig. 11.1.

more states than k and which operates exactly as S in experiments not exceeding length l, and which becomes different from S only in experiments longer than l.

Assume, for example, that we have a finite automaton A and an associated output converter (see diagram of Fig. 11.1), on which we perform experiments of length $l \leqslant 3$. It is easily seen that if $l \leqslant 3$ and the initial state is $\varkappa_1$ or $\varkappa_2$, automaton B (diagram of Fig. 11.2) generates the same output λ as A; thus, at $l \leqslant 3$, A and B do not differ. They become dissimilar only when the input consists of the fourth symbol.

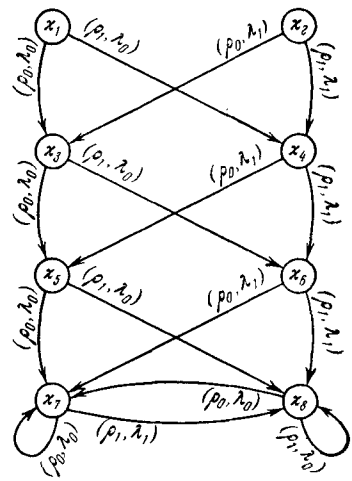


Fig. 11.2.

This argument shows that in order to experimentally determine the specific internal structure of a given automaton or s-machine one must have, in addition to the number of input symbols r, an estimate of the number k of its states. We shall assume from now on that the k and r are always known. Then we can consider the following experimental problems:

a) Determination of equivalence of two states of either the same, or of different s-machines.

b) Determination of equivalence of two s-machines.

c) Determination of the state diagram of an s-machine.

d) Determination of the state in which the machine was at the beginning of the experiment or, alternatively, its reduction to a specific state at the end of the experiment.

To solve these problems one must know what experiments can be carried out with the given set of s-machines (for example, whether one can perform a multiple experiment), as well as some additional data on this set (for example, this information may consist of the number of states k, as well as of the fact that all of these states are nonequivalent).

The next section shows a determination of the equivalence of states of an s-machine (Moore's theorem). Subsequent sections deal with the study of s-machines when multiple experiments are possible (Section 11.3), as well as with the case where only a simple experiment (in particular, a branching one) is possible (Section 11.4).

11.2. DETERMINATION OF EQUIVALENCE OF STATES OF s-MACHINES FROM THEIR RESPONSE TO FINITE INPUTS

Consider two equivalent states of some s-machine. By definition, the outputs in this case will coincide at any input, regardless of which of these equivalent states is the initial one. Conversely, if the initial states are nonequivalent, then there exists an input such that, starting with the qth cycle, the two outputs will differ. Here, q depends not only on the specific s-machine under consideration (its internal structure and the number of its states k), but also on the "discriminating" input sequence. Our problem consists of finding what is the minimal length of an input sequence capable of demonstrating the nonequivalence of two states of the given s-machines. It turns out that we can evaluate this length starting only with number (k) of the states of the machine. This length is given by the following theorem:

Theorem 1 (Moore's Theorem). If all k states of an s-machine N are nonequivalent, then for each pair of these states there exists an input sequence not longer than $k-1$, capable of discriminating between them.*

Consider the decomposition of the set of states of N into groups equivalent in terms of set L_s of all sequences of length s ($s = 1, 2, \ldots, n-1$). We shall prove the theorem by induction with respect to s. We shall prove that if the number of groups of states equivalent

*See [72]; see also [98] where the same theorem has been independently proven.

in terms of L_s is m_s, then the number of groups equivalent in terms of L_{s+1} is not less than $m_s + 1$ (that is, $m_{s+1} > m_s + 1$).

If $s = 1$, that is, all input sequences consist of one symbol, then we can decompose the set of states of N into at least two groups of states equivalent in terms of L_1. Indeed, if *all* states of N were equivalent in terms of L_1, they would also be equivalent in terms of set E of all possible sequences (since in this case the output of the machine would be governed only by its input). However, this is not the case here because N has no states equivalent in terms of E. Consequently, $m_1 \geqslant 2$.

Now select two states $\varkappa_i$ and $\varkappa_j$ which are equivalent in terms of L_s. By our specification of N, $\varkappa_i$ and $\varkappa_j$ are nonequivalent states; therefore there must exist some input sequence capable of discriminating between them, but this sequence does not belong to L_s. Let the minimum length of this sequence be q (where $q > 1$). Then the first $q - (s + 1)$ symbols of this sequence will cause the machine to shift from the states $\varkappa_i$ and $\varkappa_j$ to states $\varkappa_g$ and $\varkappa_h$, respectively, which are also equivalent in terms of L_s. In fact, since q is the minimum length of the discriminating input, then, if the initial states are $\varkappa_i$ and $\varkappa_j$, the respective outputs must coincide from the $(q - s - 1)$-th to the $(q - 1)$-th machine cycle inclusively. For this reason, the outputs will coincide from the time $q - s - 1$ (at which the machine will be in states $\varkappa_g$ and $\varkappa_h$, respectively). However, we also know *a priori* that states $\varkappa_g$ and $\varkappa_h$ can be discriminated by an input of length $s + 1$, since the $(s + 1)$-th cycle after the initial states $\varkappa_g$ and $\varkappa_h$ is the qth cycle after the initial states $\varkappa_i$ and $\varkappa_j$, and q was *a priori* chosen in such a way that the outputs in the qth cycle will be different. Consequently, states $\varkappa_g$ and $\varkappa_h$ are *a priori* known to belong to different groups which are equivalent in terms of L_{s+1}.

Let us note now that if two states $\varkappa_g$ and $\varkappa_h$, which are equivalent in terms of L_s, are nonequivalent in terms of L_{s+1}, then the group of states equivalent in terms of L_s to which $\varkappa_g$ and $\varkappa_h$ belong may be decomposed into at least two groups equivalent in terms of L_{s+1}. This proves that $m_{s+1} \geqslant m_s + 1$.

It follows from this inequality and the inequality $m_1 \geqslant 2$ proved above that there always exists a $q^* \leqslant k - 1$ such that the number of groups of states equivalent in terms of L_{q^*} is exactly equal to k; that is, any two states of N are nonequivalent in terms of L_{q^*}. But this means that for each pair of states of machine N there exists an input sequence from L_{q^*} no longer than $q^* = k - 1$ at which the outputs do not coincide. This proves Moore's theorem.

Now, for a few notes in connection with this theorem.

Note 1. If the given automaton is associated with an output converter, and if we know not only the number k of nonequivalent states

but also the number l of symbols in the table of the output converter, then, instead of the estimate of $(k-1)$, the estimate of $q^* = k - l + 1$ will apply. The proof of this statement follows the above proof of Moore's theorem word for word, the only change being that in this case the inequality $m_1 \geqslant l$ applies instead of the inequality $m_1 \geqslant 2$; that is, the number of groups of states equivalent in terms of L_1 cannot be less than l. If, however, l is not known *a priori*, then one uses the "worst" case in the estimate, that is, the case when $l = 2$ and $k - l + 1 = k - 1$.

Note 2. We can easily show that the estimate of the length of the sequence capable of discriminating between nonequivalent states and given by Theorem 1 is exact in the sense that this length cannot be shortened regardless of which s-machine with k nonequivalent states is used. This follows from the fact that for each k we may devise machines in which two nonequivalent states are *a priori* known to be indistinguishable if the "discriminating" input is shorter than $k - 1$.

Example. Consider a finite automaton (Table 11.1) associated with an output converter (Table 11.2) whose state diagram is shown in Fig. 11.3. It is easily seen that to establish nonequivalence of

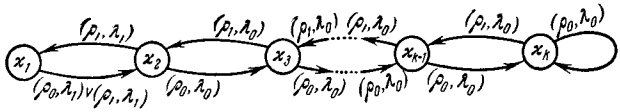


Fig. 11.3.

Table 11.1

z \ ρ	ρ_0	ρ_1
z_1	z_2	z_2
z_2	z_3	z_1
z_3	z_4	z_2
...	...	...
z_{k-1}	z_k	z_{k-2}
z_k	z_k	z_{k-1}

Table 11.2

z	λ
z_1	λ_1
z_2	λ_0
z_3	λ_0
...	...
z_{k-1}	λ_0
z_k	λ_0

states $\varkappa_{k-1}$ and $\varkappa_k$, the input sequence cannot be shorter than $k-1$; this is because the machine starting from these states, will generate only λ_0 at any input shorter than $k-1$. If, however, the output of machine in state $\varkappa_2$ were λ_2, then the nonequivalence of $\varkappa_{k-1}$ and $\varkappa_k$ could be established by a sequence only $k-2$ long (that is, $k-l+1$, where $l=3$).

Note 3. The arguments used in the proof of Theorem 1 may also be used for proving the equivalence (or nonequivalence) of two states of a single s-machine of known structure (that is, a machine with known state diagram, or tables of the automaton and converter). In that proof the machine states must first be divided into groups equivalent in terms of L_1; each of the groups so obtained must then be subdivided into groups equivalent in terms of L_2, and so on, until the two states under consideration appear in different groups. If this does not occur by the $(k-1)$-th step (that is, after all the states have been subdivided into groups equivalent in terms of L_{k-1}), then by virtue of Theorem 1 the two states under consideration are equivalent. We used an essentially similar argument in Section 9.4.

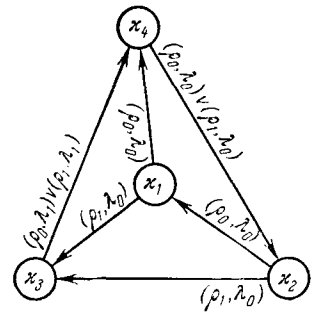


Fig. 11.4.

Note 4. Although any two nonequivalent states can be distinguished by an input not longer than $k-1$, this discriminating input does, in general, vary in length with different pairs of (nonequivalent) states. Thus, in general, there is no single finite input sequence capable of discriminating any one of the states from all the others.

Example. Consider a finite automaton (Table 11.3) associated with an output converter (Table 11.4), shown in Fig. 11.4. Here one

Table 11.3

$\varkappa$ \ ρ	ρ_0	ρ_1
$\varkappa_1$	$\varkappa_4$	$\varkappa_3$
$\varkappa_2$	$\varkappa_1$	$\varkappa_3$
$\varkappa_3$	$\varkappa_4$	$\varkappa_4$
$\varkappa_4$	$\varkappa_2$	$\varkappa_2$

Table 11.4

$\varkappa$	λ
$\varkappa_1$	λ_0
$\varkappa_2$	λ_0
$\varkappa_3$	λ_0
$\varkappa_4$	λ_1

can distinguish between states $\varkappa_1$ and $\varkappa_2$ if the input sequence starts with ρ_0; but discrimination between states $\varkappa_1$ and $\varkappa_3$ requires that the input starts with ρ_1.

Note 5 (which is the direct consequence of *Note 4*). There exists no simple experiment which can tell, even if the state diagram of the s-machine is available, what the state of the machine was at start of the test. Indeed, it has been shown that there is no finite experiment capable of distinguishing between a given initial state and all the others. And if we carry out some experiment capable of distinguishing a given state $\varkappa_i$ from some subset S of the set K of all the states of this s-machine, the very experiment will automatically shift this s-machine out of the state $\varkappa_i$, and thus we will be unable to determine in which of the states of the subset $K - S$ it had been initially.

It can, of course, be easily seen that when the machine has no equivalent states and we can perform a multiple experiment (that is, we have several identical s-machines, or a machine with reset), we can always find the initial state.

Note 6. Theorem 1 gives an estimate of the length of experiment capable of determining the nonequivalence of the states of two s-machines, having k_1 and k_2 internal states, respectively. This can be done by regarding these states as states of a single s-machine obtained by simple union* of these two s-machines. After this union, the nonequivalence of the two states may be established by an experiment not longer than $q^* = k_1 + k_2 - 1$.

The nonequivalence of states of two different automata with output converters can be established by an experiment (see *Note 1*) not longer than $q^* = k_1 + k_2 - l + 1$.

If $k_1 = k_2$, then the respective estimates become:
For the two s-machines $q^* = 2k - 1$, and for the two automata with converters $q^* = 2k - l + 1$. The next example will show that these values cannot be improved upon.

Example. Figure 11.5 shows the diagrams of two finite automata without output converters, where the number of output symbols $l = 2$. It is readily seen that an experiment establishing the nonequivalence of states $\varkappa_1'$ and $\varkappa_1''$ cannot be shorter than 7 (for example, the input sequence could be $\rho(t) = \rho_0\rho_0\rho_0\rho_0\rho_0\rho_1\rho_1\rho_1$). If, however, states $\varkappa_2'$ and $\varkappa_2''$ of these automata were associated with a new output λ_2, that is, if l were 3, then experiment of length 6 is sufficient. Similar examples can be devised for k.

*The state diagram of the combined machine is a simple union of the diagrams of the component machines.

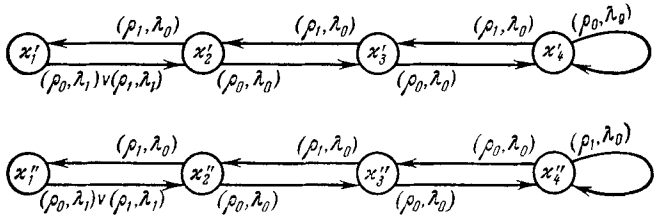


Fig. 11.5.

11.3. MULTIPLE EXPERIMENTS ON SEQUENTIAL MACHINES

The multiple experiment requires several identical s-machines or a machine with reset. In these experiments we consider only those states which the machine may attain in a finite number of steps, starting from state $\varkappa^0$.

A sequential machine is said to be $\varkappa^0$-*connected* if it has a diagram such that for each state $\varkappa_i$ $(i = 1, 2, \ldots, k)$ there exists an input capable of shifting this machine from its initial state $\varkappa^0$ to state $\varkappa_i$. It is quite obvious that our discussion should not go beyond $\varkappa^0$-connected machines: if the machine were not $\varkappa^0$-linked, then our multiple experiment will permit us to study only that section of it which is $\varkappa^0$-connected. For that reason, we shall discuss only $\varkappa^0$-connected machines with reset. With such machines there is no problem of machine states at the beginning or the end of the experiment, and the only problems which can be considered are those of the equivalence of two s-machines and of determining the diagram of the machine.

Let us first discuss the equivalence problem. It is obvious that the determination of equivalence of two $\varkappa^0$-connected s-machines may be reduced to a determination of equivalence of the two states $\varkappa^0$ in these two machines. But we have shown in the Section 11.2 that the nonequivalence of states of two such s-machines can be proven by an experiment not longer than $2k - 1$ or, in the case of two automata with converters, by an experiment not longer than $2k - l + 1$. Thus the multiple experiment can discriminate a specific $\varkappa^0$-connected, s-machine from the whole class of $\varkappa^0$-connected machines which are nonequivalent to it and whose diagrams are known. From this follows a technique for solving the second problem, that of constructing the diagram of this $\varkappa^0$-connected, s-machine, the algorithm of which is as follows:

1. We preform all the possible experiments of length $2k-1$ on the machine (a total of r^{2k-1} experiments). We record the results in the form of tables (tapes), leaving blank the table entries corresponding to the states of the s-machine.

2. We assign some number i $(1 \leqslant i \leqslant k)$ to the initial state of the machine and substitute this number into the corresponding positions of the table.

3. After the first step of the experiment, the machine will be in one of states $\varkappa^1$, of which there can be no more than r. We use all the inputs of length $k-1$ to find out whether there are any equivalents among the states $\varkappa^0$ and $\varkappa^1$. We assign arbitrary and different numbers i and j $(1 \leqslant i,\ j \leqslant k)$ to all the states $\varkappa^1$ which are nonequivalent to each other and to state $\varkappa^0$. Those states that are equivalent are coded by the same number. Let the number of different states $\varkappa^1$ be r_1.

4. From each of the states so coded no more than r new states $\varkappa^2$ may be reached in one step, so that the total number of states $\varkappa^2$ cannot exceed $r_1 r$. We use all the possible input sequences of length $k-1$ to ascertain whether there are equivalents among states $\varkappa^0$, $\varkappa^1$ and $\varkappa^2$. We assign numbers to states $\varkappa^2$ in the same way as we have coded states $\varkappa^1$.

5. We continue this process until we find k states nonequivalent to each other. It is obvious that this number will be reached in less than $2k-1$ steps (that is, we need not scan all the experimentally derived tapes).

6. We construct a state diagram, a basic table, or an interconnection matrix in accordance with the experimental results.

Note. Because the output of a finite automaton with an output converter is governed by its state $\varkappa$ and is independent of the input ρ supplied at the given time, we need only r^{2k-l} experiments of length $2k-l+1$ (instead of r^{2k-l+1}) to derive the diagram of this automaton. In addition, the last input symbol in each experiment may be arbitrary, for example, the same one for all experiments.

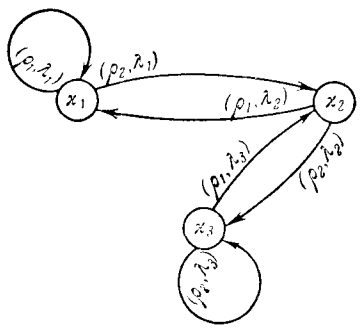


Fig. 11.6.

Example. Suppose we know that a certain automaton associated with an output converter has $k=3$ nonequivalent states, $r=2$ inputs, and $l=3$ output symbols. Then experiments of length $2k-l+1=4$, performed in this automaton as per the above algorithm, allow us to enter the states into the

table (see the symbols in parentheses of Table 11.5) and to construct the state diagram of the automaton (Fig. 11.6).

Table 11.5

t	0	1	2	3
p	p_1	p_1	p_1	p_1
$\varkappa$	$(\varkappa_1)$	$(\varkappa_1)$	$(\varkappa_1)$	$(\varkappa_1)$
λ	λ_1	λ_1	λ_1	λ_1

t	0	1	2	3
p	p_2	p_2	p_1	p_1
$\varkappa$	$(\varkappa_1)$	$(\varkappa_2)$	$(\varkappa_3)$	$(\varkappa_2)$
λ	λ_1	λ_2	λ_3	λ_2

t	0	1	2	3
p	p_1	p_1	p_2	p_1
$\varkappa$	$(\varkappa_1)$	$(\varkappa_1)$	$(\varkappa_1)$	$(\varkappa_2)$
λ	λ_1	λ_1	λ_1	λ_2

t	0	1	2	3
p	p_2	p_1	p_1	p_1
$\varkappa$	$(\varkappa_1)$	$(\varkappa_2)$	$(\varkappa_1)$	$(\varkappa_1)$
λ	λ_1	λ_2	λ_1	λ_1

t	0	1	2	3
p	p_1	p_2	p_2	p_1
$\varkappa$	$(\varkappa_1)$	$(\varkappa_1)$	$(\varkappa_2)$	$(\varkappa_3)$
λ	λ_1	λ_1	λ_2	λ_3

t	0	1	2	3
p	p_2	p_2	p_2	p_1
$\varkappa$	$(\varkappa_1)$	$(\varkappa_2)$	$(\varkappa_3)$	$(\varkappa_3)$
λ	λ_1	λ_2	λ_3	λ_3

t	0	1	2	3
p	p_2	p_1	p_2	p_1
$\varkappa$	$(\varkappa_1)$	$(\varkappa_2)$	$(\varkappa_1)$	$(\varkappa_2)$
λ	λ_1	λ_2	λ_1	λ_2

t	0	1	2	3
p	p_1	p_2	p_1	p_1
$\varkappa$	$(\varkappa_1)$	$(\varkappa_1)$	$(\varkappa_2)$	$(\varkappa_1)$
λ	λ_1	λ_1	λ_2	λ_1

11.4. SIMPLE EXPERIMENTS ON SEQUENTIAL MACHINES

If multiple experiments cannot be performed, we can study s-machines by means of simple experiments.

The problem of discerning nonequivalence and determining the internal structure of s-machines by simple experiments has been solved only for the case of the set of machines in which all the states are nonequivalent. Such a set is that of differing strongly connected machines. A sequential machine is said to be *strongly connected* if for each pair $\varkappa_i$ and $\varkappa_j$ of its states there exists an input sequence capable of shifting it from state $\varkappa_i$ to state $\varkappa_j$.

Because the class of strongly connected machines is narrower than that of $\varkappa^0$-connected machines, it follows from Section 11.3 that *two strongly connected s-machines are equivalent if at least two states of these machines are equivalent.* Therefore all the states of all the machines of a set consisting of differing strongly connected machines are nonequivalent.

As stated in Note 5 to Theorem 1 (see Section 11.2), in general there is no simple experiment capable of distinguishing an initial $\varkappa^0$ of an s-machine from all the other states which are nonequivalent to it. For this reason we would want to find a simple experiment which would shift the machine into a state which could be uniquely specified; in other words, we desire an experiment in which there exists a unique correspondence between the result and the last state of the experiment $\varkappa^p$ (the state that corresponds to the last input symbol being tested). That an experiment exists, and that the entire class of s-machines may be subjected to it is proved by Theorem 2, which also provides an estimate of its length.

Theorem 2 (the Moore-Karatsuba Theorem). The last state of a given s-machine with k nonequivalent internal states is obtainable from an experiment not longer than $\frac{k(k-1)}{2}$ *or, in the case of a finite automaton, not longer than* $\frac{(k-1)(k-2)}{2}+1$.

Proof. Assume that the state diagram of the s-machine is given. We shall try to find the input sequence discriminating the last state of this machine in the form of a series of *consecutive* sequences (that is, experiments) a_s $(s = 1, 2, \ldots)$. These sequences shall be such that the set T_s of the possible states* occurring after the input of

*In papers [72], [25], [41], [59], and [60], the set T_s is called the set of associated states. Let us emphasize that T_s is the set of those states which occur after the input of sequence a_s, and is is not the set of possible states which govern the last observed output symbol (and thus determine the decomposition of the set of all the states into groups equivalent in terms of a_s).

the last symbol of experiment a_s (these states are therefore the possible initial states for the next experiment a_{s+1}) will contain not more than $k - s$ elements. If the s-machine is an automaton, then such states would include at least two states which can be distinguished by an experiment of length 1, that is, by any input symbol.

This condition is satisfied before the beginning of the experiment, when $s = 0$. Now we shall prove that if this condition is satisfied for a_s, then there exists an a_{s+1} for which it also holds. The initial machine state for the experiment a_{s+1} must belong to the set of states T_s. Using Theorem 1 and the set of arguments used in its proof, we find that the elements of T_s (there can be no more than $k - s$ such elements, in accordance with the condition of Theorem 2) may belong to:

a) at least two groups of states equivalent in terms of set L_{s+1} of all experiments not longer than $s + 1$ (there are at least $s + 2$ such groups; see Theorem 1); and

b) at least three groups of states equivalent in terms of set L_{s+2} of all experiments not longer than $s + 2$ (there are at least $s + 3$ such groups).

Consequently, for any s-machine there will always be, among the $k - s$ states of set T_s, a pair of states which can be distinguished by an $s + 1$ long experiment a_{s+1}. For this reason set T_{s+1} will contain at least one element less than T_s, that is, it will contain not more than $k - (s + 1)$ states.

If, however, our machine is an automaton, then, by virtue of (b), set T_s will always contain a pair of states that can be distinguished by an $s + 2$ long experiment, in which the sequence of the first $s + 1$ symbols is regarded as the experiment a_{s+1}. The theorem stipulates that in an automaton there are at least two states of T_s that are distinguishable by any input symbol. For this reason, the first symbol of the experiment a_{s+1} will discriminate between these symbols. Consequently, set T_{s+1} will contain at least one element less than set T_s, that is, it will contain $k - (s + 1)$ states, at least two of which, by virtue of our choice of experiment a_{s+1}, will be discriminated by any input symbol.

Since the theorem holds for s and $s + 1$, it follows by induction that it will hold for any positive integral s; thus T_{k-2} will contain not more than two states which, in the case of an s-machine, can be distinguished by an experiment a_{k-1} not longer than $k - 1$ or, in the case of an automaton, by an input symbol (that is, by an experiment of length 1).

Thus, none of the experiments a_s is longer than s, and the last state of the s-machine may be determined by an experiment not

longer than

$$\tilde{q} = \sum_{s=1}^{k-1} s = \frac{k(k-1)}{2} \tag{11.1}$$

while the length of a similar experiment required in the case of a finite automaton is

$$\tilde{q} = \sum_{s=1}^{k-2} s + 1 = \frac{(k-1)(k-2)}{2} + 1 . \tag{11.2}$$

Note 1. The two examples given below show that the above-calculated required experimental length cannot be shortened.

Example 1. In order to distinguish the last state of an automaton with diagram of Fig. 11.7, we require an experiment $\rho_1\rho_2\rho_1\rho_1$

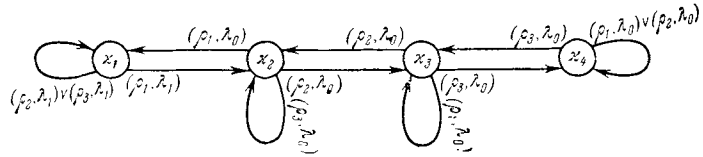


Fig. 11.7.

of length 4, that is, of length exactly equal to $\frac{(k-1)(k-2)}{2} + 1$.

Example 2. In order to distinguish the last state of an s-machine with diagram of Fig. 11.8, we require an experiment $\rho_3\rho_1\rho_3\rho_2\rho_1\rho_3$ of the length $\frac{k(k-1)}{2} = 6$.

It is readily shown that no shorter experiment will accomplish this in either example. The technique for devising similar examples for any k is obvious.

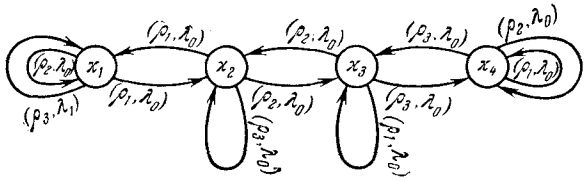


Fig. 11.8.

Note 2. If the output alphabet is taken to contain l symbols then, by a similar reasoning, we arrive at an estimate of the length of experiment determining the last state of an automaton with an output converter:

$$\tilde{q} = \frac{(k-l+1)(k-l)}{2} + 1. \tag{11.3}$$

In the case of automaton without a converter ($k = l$), we obtain the obvious estimate of 1.

Note 3. The experiment determining the last state is shorter in cases where the initial states are known to be a subset of the entire set of states k. If the total number of possible initial states is $1 \leqslant \nu \leqslant k$, and among those states there are states which can be distinguished by any input symbol, then we can prove by reasoning similar to that used in the proof of Theorem 2 that the length of an experiment discriminating the last state of an automaton with a converter must be

$$\tilde{q} = \frac{\nu - 2}{2}(2k - 2l - \nu + 3) + 1. \tag{11.4}$$

When none of the ν initial states of an automaton with a converter is distinguishable by an experiment of length 1, then the length of the required experiment will be

$$\tilde{q} = \frac{(\nu - 1)(2k - 2l - \nu + 4)}{2}. \tag{11.5}$$

Example 3. If the only initial states of the automaton of Fig. 11.7 are $\varkappa_3$ and $\varkappa_4$, then the last state will be distinguished by an experiment $\rho_2\rho_1\rho_2$ of length 3, that is, of length exactly equal to $\frac{(\nu - 1)(2k - 2l - \nu + 4)}{2}$. If, however, only $\varkappa_1$ and $\varkappa_3$ are initial states then the last state will be distinguished by any input symbol, that is, by an experiment of length

$$\frac{(\nu - 2)}{2}(2k - 2l - \nu + 3) + 1 = 1.$$

Note 4. In discussing the shortest possible experiments, we should note that if T_s contains less than $k - s$ elements (for example, $k - s - m$ elements), then reasoning similar to that used in proving Theorem 2 will show that the length of the sequence which follows a_s is not $s + 1$, but $s + m + 1$. However, in this case the total length of an experiment shifting the machine into a specific last state is shorter because sequences ranging in length from $s + 1$ to $s + m$ drop out.

We shall now illustrate a regular technique for finding the shortest experiment giving the last state of an automaton with a converter. This procedure follows directly from Theorem 2.

Example 4. Consider a finite automaton associated with an output converter whose diagram is that of Fig. 11.9, the basic table is Table 11.6, and the converter Table is 11.7. Table 11.8 shows the procedure for finding the shortest experiment for determining the last state of this automaton.

Table 11.6

x \ p	p_0	p_1
x_1	x_5	x_4
x_2	x_3	x_2
x_3	x_4	x_3
x_4	x_5	x_4
x_5	x_6	x_2
x_6	x_7	x_3
x_7	x_2	x_1

Table 11.7

x	λ
x_1	λ_1
x_2	λ_0
x_3	λ_0
x_4	λ_0
x_5	λ_0
x_6	λ_0
x_7	λ_0

Table 11.8

s	a_s	T_s	Number of elements in T_s
0	»	$\{x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$	$k = 7$
1	$a_1 = p_1 \not{p}_1$	$\{x_4\}$ $\{x_2, x_3, x_4, x_1\}$	$k-s-m = 7-1-2=4$
4	$a_4 = p_0p_0p_0p_1\not{p}_1$	$\{x_1\}$ $\{x_2, x_3, x_1\}$	$k-s=7-4=3$
5	$a_5 = p_0p_0p_0p_0p_1(\not{p}_0 \vee \not{p}_1)$ $p_0 \vee p_1$	$\{x_2\}$ $\{x_3, x_1\}$	$k-s=7-5=2$

Since we are considering an automaton, then, in accordance with the proof of Theorem 2 we select for each s those input sequences of length $s + 1$ which can discriminate any two states of set T_s, and use their first s symbols (the last, $s + 1$ symbols of the Table 11.8 are crossed out). As shown in Note 4 to Theorem 2, in this example we can go directly from $s = 1$ to $s = 4$, dispensing with $s = 2$ and $s = 3$. We can distinguish the two penultimate states x_3 and x_1 by means of any single input symbol. For this reason, the overall experiment giving the final state of our automaton will be $p_1p_0p_0p_0p_1p_0p_0p_0p_0p_1p_0$ or $p_1p_0p_0p_0p_1p_0p_0p_0p_0p_1p_1$ its length being 11.

Note 5. Another consequence of Theorem 2 is that if *a set* $\{S\}$ *consisting of machines* S_i *with* k_i *internal states (where* $i = 1, 2, \ldots, N$*) is given (that is, the diagram of each machine of this set is known), and if none of the states of these machines are equivalent*

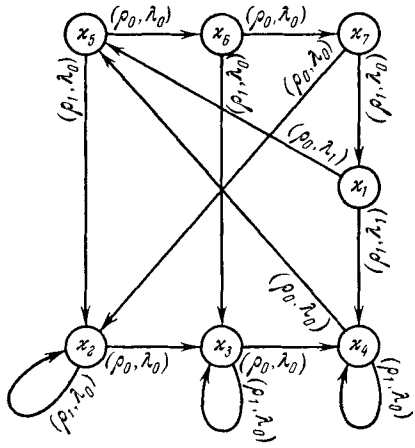


Fig. 11.9.

to each other, then there exists a simple branching experiment which permits us to distinguish any element S_x of set $\{S\}$ from all the other elements of that set.

There are two lines of approach to devising this experiment and to estimating its length. First, we can devise a branching experiment consisting of a series of sequences, each of which will shift each of the machines of the set into some known final state. These sequences are then followed by sequences discriminating these final states from each other. This is the approach proposed by Moore [72]. Thus, if we deal with an s-machine set $\{S\}$ consisting of S_i elements with internal states (where $i = 1, 2, \ldots, N$), then the length of such an experiment will be [from (11.3) and (11.7)]

$$q^*_{\Pi} = \sum_{i=1}^{N} \frac{k_i (k_i - 1)}{2} + (N-1)(k_{1\max} + k_{2\max} - 1), \tag{11.6}$$

where $k_{1\max}$ and $k_{2\max}$ are the maximal numbers of states.

If all machines have the same number of internal states, the length of the experiment will obviously be

$$q^*_{\Pi} = N\frac{k(k-1)}{2} + (N-1)(2k-1). \tag{11.7}$$

In the case of finite automata with converters we get from (11.4) and (11.9)

$$q^*_A = \sum_{i=1}^{N} \left[\frac{(k_i - l)(k_i - l + 1)}{2} + 1 \right] + \\ + (N-1)(k_{1\max} + k_{2\max} - l + 1) \tag{11.8}$$

or, when all machines have the same number of states

$$q_A^* = N\left[\frac{(k-l)(k-l+1)}{2} + 1\right] + (N-1)(2k-l+1). \qquad (11.9)$$

Finally, if we are dealing with an automaton not associated with a converter, that is, if $k = l$, then any input symbol will determine the state in which each automaton is (see Note 2) and the length of the experiment permitting the discrimination of one of the N automata will be

$$q_{\Sigma A} = 1 + (N-1)(k+1). \qquad (11.10)$$

If we have one real machine S_x of the given set $\{S\}$ and the state diagrams of all the machines of the set, then we can devise such an experiment in the following manner:

1) From their state diagrams we find all the possible experiments determining the final states of all the machines of set $\{S\}$. Assume that for machines $S_1, S_2, \ldots, S_N$ we have experiments $a_1, a_2, \ldots, a_M$ $(M \leqslant N)$. Even though all these machines are nonequivalent, each of these experiments may give identical results (the results can depend on the initial state of a machine). Thus, each of these experiments can produce identical results in the machine whose final state the experiment uniquely determines, as well as in the other machines of the set.

2) We perform a mental experiment a_1 on machine S_1 consecutively, starting from all of its possible initial states. We also perform the same experiment of the real machine S_x under investigation. If at any of the initial states the experimental results for S_1 coincide with those for S_x, we have determined a final state of S_1 which may also possibly be a final state of S_x. If the results of the experiment a_1 with S_1 differ from those of the same experiment with S_x at all possible initial states of S_1, we eliminate this machine from further consideration.

If the same experiment a_1 determines the final states of several machines of set $\{S\}$, for example, those of machines $S_{\alpha_1}, S_{\alpha_2}, \ldots, S_{\alpha_q}$, and if all (or some) of these results, at some initial states, coincide with the experimental results on S_x, we have determined the final states of these machines, which may also possibly be final states of S_x. If there is no such matching of results we eliminate these machines from further discussion.

3) We perform a mental experiment a_2 with the corresponding machine S_2 (or with machines $S_{\beta_1}, S_{\beta_2}, \ldots, S_{\beta_h}$) at all of its possible initial states. We carry out the same experiment with the real

machine S_x, as well as with machine S_1 or with those of machines $S_{\alpha_1}, S_{\alpha_2}, \ldots, S_{\alpha_q}$ which were not eliminated in (1) and (2). The initial states taken for machines S_1 or $S_{\alpha_1}, S_{\alpha_2}, \ldots, S_{\alpha_q}$ are those determined by their final states and the last symbol of experiment a_1. We then drop those of machines S_1 and S_2 (or $S_{\alpha_1}, S_{\alpha_2}, \ldots, S_{\alpha_q}$ or $S_{\beta_1}, S_{\beta_2}, \ldots, S_{\beta_h}$) for which the results of experiment a_2 do not coincide with the results of the experiment with the real machine S_x, and thus establish the final states of the remaining machines.

We continue in the same manner with other experiments until we have performed all the experiments a_i which determine the final states of all the machines of set $\{S\}$. Our result may then show that S_x can belong to a subset $\{\bar{S}\} \subseteq \{S\}$ of machines reduced to some definite states.

If the given set contains automata without converters, then, as we have already indicated, any input symbol will yield the final states of all the automata.

4) We find from the state diagrams an experiment b_1, discriminating between the states of any two machines S_m and S_n of $\{\bar{S}\}$. We then perform this experiment on S_m and S_n and on the real machine S_x. This eliminates either both of these machines, or one of them. We note the final state of the remaining machine.

5) We select another one or two machines from $\{\bar{S}\}$ and perform on it (or them) the same experiment b_1. If the result(s) match that obtained in (4) on S_x, we note the final state of the remaining machine (or machines).

6) We find an experiment discriminating between the state of the machine remaining in (4) and that of the machine(s) remaining in (5). We perform experiment b_2 with this pair of machines and with the machines S_x, as indicated in (4). We then follow the instructions of the algorithm (1-5) until all the machines set $\{\bar{S}\}$ but one are eliminated, the state diagram of the remaining machine being that of the real machine S_x.

Example 5. Assume we are given the diagrams of three automata (Fig. 11.10) and we are required to find out which of these diagrams corresponds to that of some real automaton whose diagram is unknown but which is available for experimentation.

Assume that real automaton happens to be S_3 in initial state $\varkappa_{23}$. Then the input of *any* symbol (for example, ρ_2) to this automaton immediately shows the possible initial states ($\varkappa_{21}$, $\varkappa_{22}$, and $\varkappa_{23}$, in this case) of this machine. Then a further input of a four-symbol sequence $\rho_2\rho_1\rho_1\rho_2$ enables us to distinguish between the states $\varkappa_{11}$ and $\varkappa_{12}$ (of the three possible states $\varkappa_{11}$, $\varkappa_{12}$, and $\varkappa_{13}$ into which the machine could be shifted by the input of ρ_2). Finally, the sequence $\rho_2\rho_1\rho_2\rho_2$

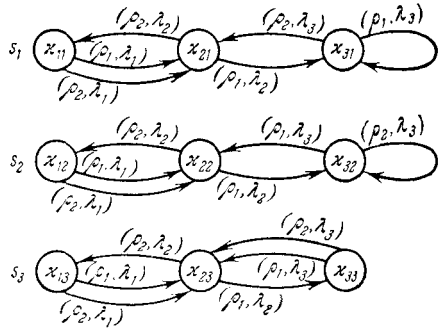


Fig. 11.10.

enables us to distinguish between states $\varkappa_{12}$ and $\varkappa_{13}$ which the machine can assume after the input of the first five symbols. Thus the entire branching experiment $\rho_2\rho_2\rho_1\rho_1\rho_2\rho_2\rho_1\rho_2\rho_2$ enabling us to distinguish one automaton from a given set will have a length of 9; that is, will exactly equal the result of expression (11.10). It is readily seen that we could have used a shorter experiment, for example, $\rho_1\rho_2\rho_1\rho_1\rho_2$, to accomplish the same purpose. But in this case we would have to carefully select all the sequences comprising the entire experiment (that is, the first step ρ_1 as well as the sequences $\rho_2\rho_1$ and $\rho_1\rho_2$ discriminating between the states), inspecting beforehand all the possible final states which can be arrived at from all possible initial states, and this would greatly complicate the algorithm of the experiment.

One can also find out the length of a simple nonbranching experiment which would enable us to distinguish one specific machine from a given set $\{S_i\}$ in which all states are nonequivalent to one another. This could be obtained by another method, starting with the simple union of elements of the given set (see the footnote on p. 290). An experiment which would determine the final state of such a combined machine at all possible initial states, would obviously enable us to distinguish any machine of the set.

In accordance with Note 6 to Theorem 1, any two states of such a combined machine can be distinguished by an experiment not longer than $k_{m_1} + k_{m_2} - 1$ (or $k_{m_1} + k_{m_2} - l + 1$ in the case of automata associated with converters), where k_{m_1} and k_{m_2} are the largest of all k_i. For this reason, the experiment determining the final state of a combined machine will consist of sequences whose estimated length increases from 1 to $k_{m_1} + k_{m_2} - 1$ (or $k_{m_1} + k_{m_2} - l + 1$ for automata with converters), and then remains constant.

Using reasoning analogous to that employed in the proof of Theorem 2, we obtain the following estimates: for a set of machines,

each with the same number of states

$$q_{U\Pi} = k(2k-1)(N-1), \tag{11.11}$$

for automata associated with converters

$$q_{UA} = (2k-l+1)\left(Nk-k-\frac{l}{2}\right)+1, \tag{11.12}$$

for automata without converters, or at $k = l$

$$q_{UA} = k(k+1)\left(N-\frac{3}{2}\right)+1. \tag{11.13}$$

It can be shown that the estimates (11.11) - (11.13) for the length of a nonbranching experiment, applicable to the union of all the machines of the set, are usually not as good as the estimates (11.7), (11.9), and (11.10) for the length of a nonbranching experiment obtained by shifting each machine of the set into some specified state and then comparing those states. Moreover, the second method is much more complicated than the first because in searching for individual sequences constituting this experiment we do not deal with the individual machines of the set, but with the set as a whole.

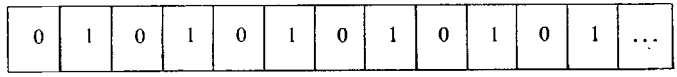
12

Algorithms

12.1. EXAMPLES OF ALGORITHMS

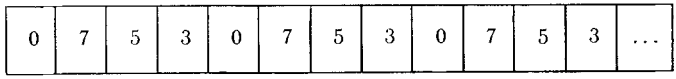
In our previous discussion we have examined many infinite sequences without stopping to think what it means "to examine" an infinite sequence. Obviously, we cannot scan it, and the only understanding of such a sequence which we can achieve derives from the analysis of its properties. We shall illustrate this concept by some examples.

Determination of a term in an infinite sequence. Consider the sequence



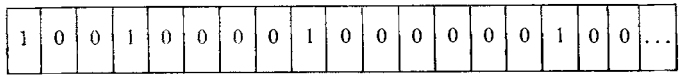
There is little that we can forecast about its further behavior. However, if we know that the symbols 0 and 1 always alternate, we can predict the term appearing in any position, because 0 always appears in an odd-numbered, and 1 in an even-numbered position.

Now consider the sequence



If we know that group 0, 7, 5, 3 is recurrent, and if we know the first term of this sequence, we can again determine any subsequent term. To find the term appearing in the nth position, we divide n by the number of terms in the recurrent group. The remainder obtained in this division indicates the position of the term in the recurrent group. If the remainder is 0, the nth term coincides with the last term of that group.

As a final example, consider the sequence



Here 1 appears only in positions whose "addresses" are squares of integers. If we know that, we know everything about the sequence. To find the nth term, we merely take the square root of n. If it is an integer, the nth term is 1. Otherwise, it is 0.

These three examples had one common "property" which enabled us to reconstitute the entire infinite sequence starting from a small segment of it. In all cases we had a "prediction procedure," that is, a procedure for determining any term, given its "address." To be more exact, in all three cases we dealt with an *algorithm* for finding the term, given the ordinal number of its position in the sequence.

An algorithm usually means a set of formal directions for obtaining the required solution. This formulation is not exact but rather expresses an intuitive concept which dates back to antiquity.*

To clarify the characteristic properties of an algorithm, let us consider some typical examples.

The Euclidean algorithm. This algorithm determines the greatest common divisor of two positive integers a and b, and may be described by the following sequence of directions:

1. Compare a and b ($a = b$, or $a < b$, or $a > b$). Go on to 2.

2. If $a = b$ then either is the greatest common divisor. Stop the computation. If $a \neq b$ go on to 3.

3. Subtract the smaller from the larger number and write down the subtrahend and the remainder. Go to the next instruction.

4. Assign symbol a to the subtrahend, and symbol b to the remainder. Return to direction 1.

The procedure is repeated until $a = b$. Then the computation is stopped.

The above set of directions, each consisting of a simple arithmetical operation (subtraction, comparison) can obviously be made more detailed, in which case the direction will be still simpler.

Algorithms which reduce the solution to arithmetical operations are termed *numerical algorithms*. Our three previous examples belonged to this class, as do formulas and procedures for solution of any class of problems, provided such formulas fully express both the nature of the operations (multiplication, subtraction, or division) and the order in which they must be performed.

A logical algorithm. Now consider an algorithm for solving a logical problem—that of finding a path in a finite labyrinth.

*The term "algorithm" itself derives from the name of the ninth century Uzbek mathematician al-Khuwārizmi, who formulated a set of formal directions, that is, rules for carrying out the four operations of arithmetic in the decimal system.

Imagine a finite system of rooms, each of which is the origin of one or more corridors. Each corridor joins two adjacent rooms, and a room may thus be connected to several other rooms. On the other hand, it may open only into a single corridor, in which case it will be a "dead-end" room. Graphically, the resulting labyrinth may be shown as a system of circles A, B, C, ..., joined by straight lines (Fig. 12.1). We shall say that room Y is accessible from room X if there exists a path leading from X to Y via intermediate corridors and rooms. This means that either X and Y are adjacent rooms or there exists a sequence of adjacent rooms X, X_1, X_2, X_3, ..., X_n, Y. If Y is accessible from X, then the path from X to Y must be simple (loopless); that is, each intermediate room is traversed only once. Thus, in the labyrinth of Fig. 12.1, one simple path from H to B is $HDCB$; but L is inaccessible from A.

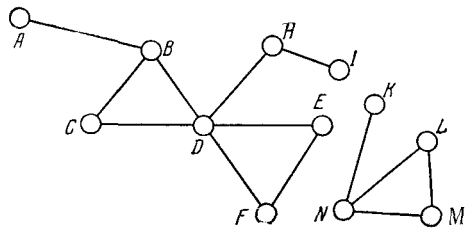


Fig. 12.1.

Suppose that we wish to ascertain whether F is accessible from A and that, if so, we wish to find a path from A to F; but, if F proves inaccessible, we wish to return to A at the end of the search. We have no map of this labyrinth and for that reason shall employ a general search procedure applicable to any labyrinth containing a finite number of rooms, with any mutual disposition of rooms A and F within it. In one such procedure, the searcher, like the mythical Theseus, holds a ball of thread, one end of which is tied down in the starting room A. In addition, the searcher can paint the corridors as he walks along them. He is thus able to distinguish those never passed before (green), those passed once (yellow), and those passed twice (red). The searcher can get from any room to an adjacent one by either of two moves:

a) He can unwind the thread. He thus stretches the thread along a "green" corridor, which then becomes "yellow."

b) He can wind the thread on the ball. He thus returns from a given room to an adjacent one, walking along a "yellow" corridor.

He picks up the thread as he walks along, and the corridor now becomes "red."*

Having arrived into any given room, the searcher may encounter any one of five possibilities:

1. This is room F, the object of the search.
2. At least two "yellow" corridors radiate from this room, that is, a thread is already stretched across it. The searcher now realizes that he has just traversed a loop.
3. At least one "green" corridor originates in this room.
4. This is the starting room A.
5. None of the above.

Now the search procedure can be specified by the following table:

Evidence in the room	*Next move*
1. Room F	Stop
2. Loop	Wind the thread
3. "Green" corridor	Unwind the thread
4. Room A	Stop
5. Fifth possibility	Wind the thread

In each room, the searcher must decide on his next move by scanning the table in numerical order and ascertaining which of the possibilities listed matches the evidence in the room. Having found the first match, he makes the necessary move without checking for other applicable possibilities. He continues to move in this fashion until the instructions are to stop.

For this procedure, we can prove the following:

1. After a finite number of moves, the searcher will stop at either A or F, regardless of the mutual dispositions of A and F.
2. If he stops at F, then he has reached his object, and the thread is now stretched along a simple path from A to F.
3. If he stops at A, then F is inaccessible.

Let us illustrate the procedure on the labyrinth of Fig. 12.1. We represent the procedure in the form of Table 12.1 and see that F is accessible from A. We write down those corridors of column 4 which remain "yellow" to the very end. These constitute the simple path from A to F.

This procedure involves an element of choice which did not exist in the previously discussed examples. Thus, two different calculators trying to find the greatest common divisor of two numbers by

*In fact, the searcher needs only two colors—green and red; this is because the "yellow" corridors contain the stretched thread.

Table 12.1

Move No.	Evidence in room determining next move	Next Move	Path (corridor) Taken	Corridor color after this move
1	Green corridor	Unwind thread	*AB*	Yellow
2	"	"	*BC*	"
3	"	"	*CD*	"
4	"	"	*DH*	"
5	"	"	*HI*	"
6	Fifth case	Wind thread	*IH*	Red
7	"	"	*HD*	"
8	Green corridor	Unwind thread	*DB*	Yellow
9	Loop	Wind thread	*BD*	Red
10	Green corridor	Unwind thread	*DF*	Yellow
11	Room *F*	STOP		

Euclid's algorithm performs operations coinciding in every detail: there is no room for individual judgment. But in the labyrinth procedure, two searchers may go from *A* to *F* via distinctly different paths.

Traditionally, the term "algorithm" is restricted to an exactly defined set of instructions. In this sense, the labyrinth search procedure is not an algorithm. To become one, it would have to be supplemented by an exact specification on what to do in the case of "green" corridors (for example, this supplemental instruction may specify that if several "green" corridors originate from the same room, the searcher must select the first one to the right of the entrance).

12.2. GENERAL PROPERTIES OF ALGORITHMS

The above examples point out some overall properties characteristic of any algorithm:

(a) *Determinancy.* The procedure is specified so clearly and precisely that there is no room for arbitrary interpretation. A procedure of this kind can be communicated to another person by a finite number of instructions. The operations described by these instructions do not depend on the whim of the operator and constitute a determinate process which is completely independent of the person carrying it out.

(b) *Generality.* An algorithm is applicable to more than just one specific problem: it is used for solving a class of problems, with the procedural instructions valid for any particular set of initial data. Thus, Euclid's algorithm is applicable to any pair of integers

$a > 0$, $b > 0$; the rules of arithmetic apply to all numbers; and the search rules hold for any finite labyrinth, however intricate.

In mathematics one considers a series of problems of a specific kind to be solved when an algorithm has been found (the finding of such algorithms is really the object of mathematics). But in the absence of an algorithm applicable to *all* problems of a given type, one is forced to devise a special procedure valid in some but not other cases. However, such a procedure is not an algorithm. For instance, there is no algorithm for finding out whether the solution of equation

$$x^n + y^n = z^n \tag{12.1}$$

is an integer at any $n = 1, 2, 3, 4, \ldots$. This problem may nevertheless be solved for particular values of n. Thus, for $n = 2$, we can easily find three numbers ($x = 3$, $y = 4$, $z = 5$) satisfying Eq. (12.1). And it may be proved that Eq. (12.1) has no integer solutions for $n = 3$. However, this proof cannot be extended to other values of n.

(c) *Efficacy.* This property, sometimes called the *directionality* of an algorithm, means that application of an algorithmic procedure to any problem of a given kind will lead to a "stop" instruction in a finite number of steps, at which point one must be able to find the required solution. Thus, no matter now intricate the (finite) labyrinth, the search algorithm must lead to a "stop" instruction in a finite number of steps. The stop will occur either at F or at A, enabling us to decide whether F is accessible or not. Again, the use of the Euclidean algorithm with any two numbers $a \geqslant 1$, $b \geqslant 1$ will sooner or later lead to a "stop" instruction, at which point one can determine the value of the greatest common divisor. However, nothing prevents us from using the Euclidean algorithm with $a \geqslant 0$ and $b \geqslant 0$, or with any pair of integers (positive or negative). There is no ambiguity at any step of the algorithm, but the procedure may not come to a stop. For example, if $a = 0$, $b = 4$, our sequence of instructions (1-4) gives the pairs 0, 4; 0, 4; 0, 4... and so on *ad infinitum*. The same will happen with the pair $a = -2$, $b = 6$.

Thus, the concept of efficacy of an algorithm naturally leads to the concept of its *range of application*. The range of application is the largest range of initial data for which the algorithm will yield results; in other words, if the problem is stated within the range of application, then the algorithm will work up the (given) conditions into a solution, after which the procedure will come to a stop; if, however, the problem conditions are outside this range, then either there will be no stop, or there will be a stop but we shall not be able to obtain a result. Thus, the range of application of the Euclidean

algorithm is the set of natural numbers {1, 2, 3, 4, ...}, and the range of application of the search algorithm is the set of all finite labyrinths.

Now, the number of individual operations which must be performed in an algorithmic procedure is not known beforehand and depends on the choice of the initial data. For this reason, an algorithm should be understood primarily as a *potentially* feasible procedure. In some specific problems stated within the range of application of the algorithm there may be no *practical* solution: the procedure may be so long that the calculator will run out of paper, ink, or time, or the computer executing the algorithm may not have a large enough memory.

Determinancy, generality, and efficacy are empirical properties. They are present in all algorithms constructed so far. However, these empirical properties are too vague and inexact to be useful in a mathematical theory of algorithms. We shall try to refine them in subsequent sections.

12.3. THE WORD PROBLEM IN ASSOCIATIVE CALCULUS

The previously described search procedure was restricted to finite, though arbitrary, labyrinths. There exists, however, a far reaching generalization of this problem which, in a sense, constitutes a search in an infinite labyrinth. This is the *word problem*. It arose first in algebra, in the theory of associative systems, and in the group theory, but the conclusions derived from it have since transcended these specialized fields.*

Let any finite system of differing symbols constitute an alphabet, and let the constituent symbols be its characters. For instance, {а, ъ, ?, 7, *} is an alphabet, whereas а, ъ, ?, 7 and * are its characters. Any finite sequence of characters from an alphabet is called a *word* of that alphabet. Thus, the three-character alphabet {*a*, *b*, *c*} will yield words *ac*, *a*, *abbca*, *bbbbb*, *bbacab*, etc. An empty word, containing no characters, is denoted by Λ.

Consider two words L and M in some alphabet A. If L is a part of M, then we say that L occurs in M. For example, the word $L = ac$ occurs in $M = bbacab$. In general, L may occur in M many times; thus *bcb* occurs twice in *abcbcbab*.

*Important contributions to its solution were made by A.A. Markov, P.S. Novikov and their students. Familiarity with this problem will help us to understand the theory of recursive functions.

We shall now describe the process of transformation of words, whereby new words are obtained from given ones. We start by setting up a finite system of substitutions allowable in a given alphabet:

$$P - Q; \quad L - M; \quad \ldots; \quad S - T,$$

where P, Q, L, M, ..., S, T are words in this alphabet, and the dashes between them denote substitution. Thus, an L—M substitution in a word R of this alphabet may be defined as follows: if L occurs one or more times in R, then any one of these occurrences may be replaced with M; conversely, if M occurs in R, then it may be replaced with L. For example, there are four possible substitutions $ab - bcb$ in $abcbcbab$. Replacement of each subword bcb with ab yields $aabcbab$ and $abcabab$, respectively, whereas the replacement of each ab yields the words $bcbcbcbab$ and $abcbcbbcb$. However, the substitution ab—bcb in $bacb$ is not allowed since neither ab nor bcb occurs in it.

The words obtained by means of allowable substitutions may be again replaced, which yields yet new words.

The aggregate of all the words in a given alphabet, together with an appropriate set of allowable substitutions, is called an *associative calculus*. To define an associative calculus, it is sufficient to define the alphabet and the set of substitutions.

Two words P_1 and P_2 of an associative calculus are said to be *adjacent* if one may be transformed into the other by a single allowable substitution. A sequence of words P, P_1, P_2, P_3, ..., Q is called a *deductive chain* leading from P to Q if every two consecutive words in this chain are adjacent. Two words P and Q are said to be *equivalent* if there exists a deductive chain leading from P to Q. The equivalence relationship is shown as $P \sim Q$. It is obvious that if $P \sim Q$, then $Q \sim P$, since the allowable substitutions may be used in either direction.

Example. Assume we have the following associative calculus:

$\{a, b, c, d, e\}$ — alphabet

$ac - ca$
$ad - da$
$bc - cb$
$bd - db$
$abac - abacc$
$eca - ae$
$edb - be$

— allowable substitutions

Here, the words *abcde* and *acbde* are adjacent, since *abcde* may be transformed into *acbde* by the substitution *bc—cb*. However, *aaabb* has no adjacent words, since none of the given substitutions may be applied to it. The word *abcde* is equivalent to *cadedb*, since there exists a deductive chain of adjacent words: *abcde, acbde, cabde, cadbe, cadedb* , derived by successive use of the third, first, fourth, and fifth of the above substitutions.

An associative calculus may be put in correspondence with an infinite labyrinth by matching a specific room of the labyrinth with a word from the alphabet. Since the number of words which may be formed from the characters of a given alphabet is infinite, it follows that the labyrinth can have an infinite number of rooms.

Two adjacent rooms of the labyrinth correspond to adjacent words. Now, if two words P and Q are equivalent, then the labyrinth room corresponding to word Q is accessible from the room corresponding to P, that is, there exists a path from P to Q.*

In each associative calculus there occurs a specific *word problem*, whereby it is required to recognize *whether two words of this calculus are equivalent or not*. This problem is identical to our problem of accessibility in a labyrinth, except that here the labyrinth is infinite. But our previous algorithm is now useless because an infinite labyrinth cannot be searched in a finite time.

Since each associative calculus contains an infinite set of different words, it involves an infinite number of problems of equivalence between two words. All these problems must involve the same procedures, and we therefore naturally think of a solution consisting of an algorithm for recognizing the equivalence (or nonequivalence) of any pair of words. We shall see in the next section whether such a solution does exist.

However, we can immediately find the algorithm for the *restricted word problem*, where one wants to know whether a given word can be transformed into another by using the allowable substitutions a maximum of k times; here k is an arbitrary but fixed number. In the previous search problem, we imposed a restriction on the labyrinth, which had to be finite. Here, however, it is the number of moves which is restricted: we must inspect all those rooms of an finite labyrinth which are separated from the starting one by

*Sometimes one uses a form of associative calculus defined by an alphabet and a system of *oriented* substitutions $P \to Q$. The arrow means that only left-to-right substitution is allowed; that is, the word P may be replaced with Q, but not the other way around. This associative calculus may be graphically represented by an infinite labyrinth in which each of the corridors is unidirectional. Obviously, the equivalence $P \sim Q$ in no way implies that $Q \sim P$ in this case.

not more than k corridors; the remainder of the labyrinth is of no interest. In terms of associative calculus this means that one determines all words adjacent to one of the given words (by substitution); then for each of the new words so derived one determines all words adjacent to it, and so on, k times in all. We finally obtain a list of all words which may be derived from the given one by using the allowable substitutions at most k times. If the second given word appears in that list, then the answer to the restricted word problem is yes; if it does not appear, the answer is no. This scanning algorithm may then be further improved by removing from it all superfluous iterations (loops).

However, the solution of the restricted word problem does not bring us nearer to the solution of our basic "unlimited" problem. Here, the length of the deductive chain (if it exists) from word P to word Q may be extremely great (or infinite). For this reason, the scanning algorithm, restricted as it is to k substitutions, generally cannot tell whether an equivalence is present or, to put it another way, when to stop the search for such an equivalence. Thus we must turn to other, more sophisticated algorithms, as we shall do in later sections.

The reader will now recognize that *logical deductive* processes other than the search problem may also be treated as associative calculi. For instance, any logical formula may be interpreted as a word in some alphabet containing the characters denoting logical variables, logical functions, and logical connectives $\vee$, $\&$, $^-$, $\to$ (,), etc. The process of *inference* may be treated as a formal word transformation similar to the substitution in associative calculus, in which an elementary act of logical inference is made to correspond to a single act of substitution. The substitutions themselves may be written as logical rules or identities, for example, $\bar{\bar{x}}—x$ (which means "double negation may be removed"), or

$$\left.\begin{array}{l}(\forall x)\,[(A_1 \vee x)\,\&\,(A_2 \vee x)] — A_1\,\&\,A_2 \\ (\exists x)\,[(A_1 \vee x)\,\&\,(A_2 \vee x)] — A_1 \vee A_2\end{array}\right\} \text{ the exclusion rule, and so on.}$$

By making such substitutions in a given premise (that is, in the wording representing such a premise), we can obtain many further inferences (conclusions).

Now, in propositional calculus there exist methods for deriving *all* the conclusions which follow from a given set of axioms. There also exists an algorithm for recognizing deducibility, that is, a procedure for checking whether a given statement follows from a given axiom or not. However, propositional calculus cannot encompass both these procedures, since it is unable to express the relationship

between one object and another within the confines of a single statement; this calls for predicate calculus—which can also be interpreted as an associative calculus. As a result, we arrive at a variant of associative calculus—the logical calculus with a given system of allowable substitutions. The problem of recognition of deducibility now becomes one of existence of a deductive chain from the word representing the premise to the word representing the inference, a problem which the reader will recognize as that of equivalence of words in associative calculus.

In the same sense, all derivations of formulas, and all mathematical computations and transformations, are processes of constructing deductive chains in the corresponding associative calculi. And we shall prove in the next section that arithmetic itself may also be treated as an associative calculus.

Given the universal applicability of associative calculus, it would be natural to postulate it as a general method for defining determinate data processing procedures, that is, algorithms. However, before we can advance this postulate, we must state precisely what we mean by an algorithm in a given alphabet.

12.4. ALGORITHMS IN AN ALPHABET *A*. MARKOV'S NORMAL ALGORITHM

By analogy with the intuitive definition of Section 12.1 ff, we could intuitively define an "algorithm in alphabet A" as follows:

Definition I. An algorithm in alphabet A is a *universally understood exact instruction* specifying a potentially realizable operation on words from A; this operation admits any word from A as the initial one, and specifies the sequence in which it is transformed into new words of this alphabet. An algorithm is *applicable* to a word P if, starting from that word and acting in accordance with this instruction, we ultimately derive a new word Q, whereupon the process comes to a halt. We then say that the algorithm processes P into Q.

For example, the following instruction satisfies our definition:

Copy a given word, beginning from the end. The word so obtained is the result. Stop.

This algorithm is an exact instruction applicable to any word.

Nevertheless, Definition I is too broad, and we shall refine the concept "algorithm in alphabet A" by means of associative calculus.

Definition II. We shall say that *an algorithm in alphabet A is a set allowable substitutions, supplemented by a universally*

*understood exact instruction which specifies the order and manner of using these allowable substitutions and the conditions at which a stop occurs.**

The following is an example of an algorithm in the sense of Definition II.

Let alphabet A contain three characters: $A = \{a, b, c\}$, and let the algorithm be defined by a set of substitutions

$$\left.\begin{array}{l} cb - cc, \\ cca - ab, \\ ab - bca \end{array}\right\}$$

and the following instructions regarding the use of these substitutions:

Starting from any word P, one scans the above set of substitutions, in the order given, seeking the first formula whose left-hand part occurs in P. If there is no such formula, the procedure comes to a halt. Otherwise, one substitutes the *right-hand part* of the *first* such formula for the *first occurrence* of its left-hand part in P; this yields a new word P_1 of alphabet A. After this, the new word P_1 is used as starting one (P in the above), and the procedure is repeated. It comes to halt upon generation of a word P_n which does not contain any of the left-hand parts of the allowable substitutions.

This set of substitutions and the instructions for its use define an algorithm in alphabet A which processes the word *babaac* into the word *bbcaaac* by means of the third substitution, at which point the procedure comes to a halt. Similarly the word *cbacacb* may be successively transformed into words *ccacacb*, *ccacacc*, *abcacc*, and *bcacacc*, at which point the procedure again comes to an end. However, the word *bcacabc* generates the recurring sequence *bcacabc*, *bcacbcac*, *bcacccac*, *bcacabc*, and so on, where no stop can occur; therefore our algorithm is not applicable to the word *bcacabc*.

This algorithm is somewhat reminiscent of the following instructions for motion in an infinite labyrinth: having arrived into a room, go to the first corridor on your right, and so on. Here, a stop will occur when a dead end is reached and, as in the algorithm, there are three possibilities: starting from any room, we can either enter a dead end corridor (compare the case of word *babaac*), or move in a loop *ad infinitum* (compare the case of word *bcacabc*), or keep going for an infinitely long time without getting trapped in a loop.

*Since an alphabet and a system of allowable substitutions define an associative calculus which, as we know, can be placed into correspondence with an infinite labyrinth, that part of the definition which relates to instructions for using the substitutions may be treated as exact instructions for moving in an infinite labyrinth.

At first glance one may conclude that Definition II is narrower than Definition I. It turns out, however, that this is not so, since for any known algorithm defined in sense I we may construct an equivalent algorithm in sense II. This, of course, does not prove that Definitions I and II are equally strong; there can be no such proof, in view of the vagueness of both definitions (for instance, both contain the undefined phrase "universally understood exact instructions"). Still, Definition II is a substantial step forward, as we shall see below.

Now let us define *equivalence of algorithms*: two algorithms A_1 and A_2 in some alphabet are equivalent if their ranges of application coincide and if they process any word from their common range of application into the same result. In other words, if algorithm A_1 is applicable to a word P, then A_2 must also be applicable to that word, and conversely; also, both algorithms must transform the word P into the same word Q. If, however, one of the algorithms is not applicable to a word B, then the other algorithm must also be inapplicable.

At this point, Definition II may be transformed into an exact mathematical definition of an algorithm by a single step first proposed by A.A. Markov. His *normal* algorithm is identical to that of Definition II except that the "universally understood" instructions are replaced by a standard, once and for all fixed, and exactly specified procedure for the use of substitutions. This normal algorithm is specified as follows: To start with, the alphabet A is defined and the set of allowable substitutions is fixed. Then some word P in A is selected, and the substitution formulas are scanned (in the order given in the set) to find a formula whose left-hand part occurs in P. If there is no such formula, the procedure comes to a halt. Otherwise the right-hand member of the first of such formulas is substituted for the first occurrence of its left-hand member in P. This yields a new word P_1 in alphabet A. After this one proceeds to the second step, which differs from the first one only in that P_1 now acts as P. Then one goes to the third analogous step, and so on, until the process comes to a halt. However, the process can be terminated in only two ways: (1) when it generates a word P_n such that none of the left-hand parts of the formulas of the substitution set occurs in it; and (2) when the word P_n is generated by the last formula of the set.

We see that the algorithm of Definition II is an "almost normal" algorithm, the only difference being that it comes to a halt in only one case (when none of the allowable substitutions is applicable), whereas in the normal algorithm there are two possible causes for a "stop" instruction.

Two normal algorithms differ only in their alphabets and their set of allowable substitutions. Again, to define a normal algorithm it is sufficient to define its alphabet and its set of substitutions.

Examples of Normal Algorithms.

Let the alphabet A and the set of allowable substitutions be

$$A = \{1, +\} \qquad \begin{array}{l} 1 + \to + 1 \\ + 1 \to 1 \\ 1 \to 1 \end{array}$$

(the arrows are a convention denoting a Markov normal algorithm, to differentiate it from the usual associative calculus).

Now let us see how this algorithm transforms the word 1 1 1 1 + + 1 1 + 1 1 1. We obtain successively the words:

1 1 1 1 + 1 1 + 1 1 1
1 1 1 + 1 1 1 + 1 1 1
1 1 + 1 1 1 1 + 1 1 1
1 + 1 1 1 1 1 + 1 1 1
+ 1 1 1 1 1 1 + 1 1 1
+ 1 1 1 1 1 + 1 1 1 1
+ 1 1 1 1 + 1 1 1 1 1
+ 1 1 1 + 1 1 1 1 1 1
+ 1 1 + 1 1 1 1 1 1 1
+ 1 + 1 1 1 1 1 1 1 1 1
+ + 1 1 1 1 1 1 1 1 1
+ 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1

The procedure comes to a halt on the use of the last substitution $1 \to 1$, which processes the word 111111111 into itself.

Now let the set of substitutions (in the same alphabet) be

$$+ \longrightarrow \lambda$$
$$1 \longrightarrow 1$$

(Λ is again an empty word). Then $11 + 111 + 1 + 11$ will be transformed as follows:

1 1 + 1 1 1 + 1 + 1 1
1 1 1 1 1 + 1 + 1 1
1 1 1 1 1 1 + 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1.

We see that both algorithms produce a sum of symbols 1; that is, they perform addition, and it not difficult to show that they are equivalent.

A third normal algorithm, equivalent to the above, is defined by a set

$$1\,+ \to +\,1,$$
$$+\,+ \to +,$$
$$+ \to \Lambda.$$

The reader is invited to verify that the normal algorithm $A = \{1, *, \vee, ?\}$

$$*11 \to \vee *1,$$
$$*1 \to \vee,$$
$$1\vee \to \vee 1?,$$
$$?\vee \to \vee ?,$$
$$?1 \to 1?,$$
$$\vee 1 \to \vee,$$
$$\vee ? \to ?,$$
$$? \to 1,$$
$$1 \to 1$$

transform each word of the form $\underbrace{1111\ldots11}_{m \text{ times}} * \underbrace{111\ldots111}_{n \text{ times}}$ into the word $\underbrace{111\ldots111}_{m\cdot n \text{ times}}$; that is, it performs a multiplication.

Markov also refined the concept of an algorithm in an alphabet by postulating that *each algorithm in an alphabet is equivalent to some normal algorithm in the same alphabet*. This is a hypothesis which cannot be rigorously proved, since it contains both the vague statement "each algorithm" and the exact concept of a "normal algorithm." This statement may be regarded as a law which has not been proved but which has been confirmed by all accumulated experience. It is supported by the fact that no one has so far succeeded in formulating an algorithm for which there is no normal algorithm equivalent to it (in the same alphabet).

Now we can return to the problem of universal definition of algorithms (see end of Section 12.3). In view of what we said above, the Markov normal algorithm seems a convenient "standard form" for defining *any* algorithm, that is, we assume that *any* algorithm may be defined as a Markov normal algorithm. This is, of course, no more than a hypothesis and, at that, much less well-founded than the Markov hypothesis discussed above, since it cannot even be expressed in exact terms. However, its intuitive meaning is obvious.

As soon as we accept this hypothesis, we have a way of rigorously proving the algorithmic unsolvability of generalized problems. For example, we can prove the algorithmic unsolvability of the word problem; that is, we can prove that there is no algorithm applicable to all associative calculi and capable of determining whether two words P and Q are equivalent. All we have to do in order to prove this is to demonstrate the existence of one associative calculus in which there is no normal algorithm for recognizing the equivalence of words. Examples of such calculi were first given by A.A. Markov (1946) and E. Post (1947). After that it became clear that *a fortiori* there can be no algorithms capable of recognizing the equivalence of words in all associative calculi.

The examples of Markov and Post were unwieldy and comprised hundreds of allowable substitutions. Later, G. S. Tseytin exhibited an associative calculus containing only seven allowable substitutions, in which the problem of word equivalence was also algorithmically unsolvable.

As an illustration we shall show one proof of algorithmic unsolvability. Let U be a normal algorithm defined in an alphabet $A = \{a_1, a_2, \ldots, a_n\}$ with the aid of a set of substitutions. In addition to the characters of alphabet A, this algorithm also uses the symbols $\rightarrow$ and ,. By assigning to these symbols new characters a_{n+1} and a_{n+2}, we can represent U by a *word* in an expanded alphabet $\tilde{A} = \{a_1, a_2, \ldots, a_{n+2}\}$. Let us now apply U to the word representing it. If algorithm U transforms this word into another one, after which there is a stop, this means that U is applicable to its own representation—the algorithm is *self-applicable*. Otherwise, the algorithm is *nonself-applicable*. Now there arises the problem of recognition of self-applicability, that is, finding out from the representation of a given algorithm whether it is self-applicable or not.

This problem would be solved by a normal algorithm V, which, upon application to any representation of a self-applicable algorithm U, would transform that representation into a word M and which would transform all representations of a nonself-applicable algorithm U into another word L. Thus the application of the recognition algorithm V would show whether U is self-applicable or not.

However, it has been proved that such a normal algorithm V does not exist (see [64]), which also proves that the problem of recognition of self-applicability is algorithmically unsolvable. The proof, by *reductio ad absurdum*, is as follows. Suppose that we do have a normal algorithm V and that it transforms each representation of a self-applicable algorithm into M and each representation of a nonself-applicable algorithm into L. Then, by some modification of the

substitution system of the algorithm V, we may devise another algorithm $\tilde{V}$ which would again transform each representation of a nonself-applicable algorithm into L but which would be inapplicable to representations of a self-applicable algorithm (because the algorithm does not come to a stop). Such an algorithm $\tilde{V}$ leads to contradictions. Indeed,

1. Suppose $\tilde{V}$ is self-applicable (there is a stop), that is, it can be applied to its own representation (which is in the form of a word). But this simply means that $\tilde{V}$ is nonself-applicable.

2. Suppose $\tilde{V}$ is nonself-applicable. Then it can be applied to its own representation (since it is applicable to any representation of a nonself-applicable algorithm). But this simply means that $\tilde{V}$ is self-applicable.

The resulting contradiction proves the algorithmic unsolvability of the problem of recognition of self-applicability.

Thus there is no *a priori* proof for the existence or nonexistence of an algorithm for a given problem. But the nonexistence of an algorithm for a class of problems merely means that this class is so broad that there is no single effective method for the solution of all the problems contained in it. Thus, even though the generalized problem of recognition of word equivalence is algorithmically unsolvable in Tseytin's associative calculus, under normal conditions we can still find a way for proving the equivalence or nonequivalence of a specific pair of words.

There is an interesting history to the problem of algorithmic unsolvability. Prior to Markov's refinement of the concept of an algorithm, mathematicians held one or the other of the following points of view:

1. Problems for which there is no algorithm are still, in principle, algorithmically solvable; the desired algorithm is unavailable simply because the existing mathematical machinery is unequal to the task of devising this algorithm. In other words, our knowledge is insufficient to solve problems we call algorithmically unsolvable, but such algorithms will be found in the future.

2. There are classes of problems for which there are no algorithms. In other words, there are problems that cannot be solved mechanically by means of reasoning and computations and that require creative thinking.

This is a very strong statement because it says to all future mathematicians: Whatever the means at your disposal may be, do not waste your time searching for nonexistent algorithms!

But how does one *prove* the nonexistence of an algorithm? So long as the definition of an algorithm comprised the phrase

"universally understood instruction" such a proof was unthinkable, because one cannot conceive of all possible "universally understood instructions" and prove that none of these is applicable.

Thus the very survival of this second viewpoint is related to the daring hypotheses on the existence of "standard forms" for defining an algorithm (such as the Markov normal algorithm), that is, hypotheses permitting the formulation of the concepts of "algorithm" and "algorithmically unsolvable problem" in exact terms.

12.5. REDUCTION OF ANY ALGORITHM TO A NUMERICAL ALGORITHM. GÖDELIZATION

The advent of computers has prompted much work in the theory of numerical algorithms with which these machines operate. In the course of this work it has been shown that any logical algorithm can be reduced to a numerical algorithm. As the methods for doing this improved, it also became clear that *all* algorithms can be reduced to numerical ones, and thus the theory of numerical algorithms (which we shall also call the theory of computable functions) became a generalized mechanism for study of all algorithmic problems.

We shall now show how any algorithmic problem can be reduced to a computation of values of an integer-valued function of integer arguments.

Assume some algorithm is applicable to a range of data. We shall represent each set of data comprised in this range by means of a *unique* nonnegative integer A_n; when we have done this, we have, instead of the original data, or collection of numerals (labels) $A_0, A_1, A_2, \ldots, A_n, \ldots$ representing these data.

Similarly, we assign a unique numeral to each of the possible solutions derived with our algorithm from the above data and thus obtain a sequence of numerals (labels) $B_0, B_1, B_2, \ldots, B_m, \ldots$ representing these solutions.

Now this labeling or *numbering* permits us to dispense with the data and solutions themselves and to operate instead on *numerals representing* these quantities. For it is fairly obvious that if we have an algorithm processing a set of data into a solution, we can also devise an algorithm processing the *numeral* A_n *denoting* these data into *the numeral* B_m representing the corresponding solution.

It is also obvious that this algorithm must be a numerical one, of the type $m = \varphi(n)$.

In general, if there exists an algorithm for solving any given problem (that is, transforming a set of data into a solution), then

there must also exist an algorithm for computing the values of the corresponding function $m = \varphi(n)$. Indeed, to find the value of $\varphi(n)$ at $n = n^*$, one can reconstitute the set of data represented by n^* from a table of values of n vs. these data; then one can employ the (existing) algorithm to find a solution for the problem. Having the value of the solution, one can go to a table of values of solutions vs. m to find the numeral m^* representing m. Consequently,

$$\varphi(n^*) = m^*.$$

Conversely, if there exists an algorithm for conputing the values of $\varphi(n)$, there must be an algorithm for solving the given problem. Indeed, one can find from the table of the data vs. n the numerical n^* representing n. Then one can compute $m^* = \varphi(n^*)$; having m^*, one can determine the value of the actual solution from another table.

Let us now present a widely used method of *numbering* (that is, unique labeling), namely, the method of Gödel. Suppose we have a number n. Then, by virtue of the fact that any composite integer can be uniquely decomposed into prime factors, we have

$$n = 2^{a_1} \cdot 3^{a_2} \cdot 5^{a_3} \cdot 7^{a_4} \cdot \ldots \cdot p_{m-1}^{a_m},$$

where $p_0 = 2$, $p_1 = 3$, $p_2 = 5$, and, in general, p_m is the mth prime number. Thus n — the Gödel number — is the product of successive primes, which are raised to powers from the set $a_1, a_2, \ldots, a_m$. Any Gödel number n is uniquely related to a specific set $a_1, a_2, \ldots, a_m$, and, conversely, each set $a_1, a_2, \ldots, a_m$ is uniquely associated with a specific number n. For example, if $n = 60$, we have: $60 = 2^2 3^1 5^1$, that is, $a_1 = 2$, $a_2 = 1$, $a_3 = 1$.

Now, Gödel numbering (or gödelization) allows us to uniquely label any sequence of m members. Consider a few examples:

1. Any pair of numbers a_1 and a_2, for which we seek the greatest common divisor q, can be assigned a unique Gödel number $n = 2^{a_1} \cdot 3^{a_2}$. Now, Euclid's algorithm reduces to the computation of $q = \varphi(n)$.

2. We want to find the sumbol in the rth position of a purely periodic sequence produced by endless repetition of the numerical sequence $a_1, a_2, \ldots, a_m$. The statement of the problem can then be associated with the Gödel number

$$n = 2^r \cdot 3^{a_1} \cdot 5^{a_2} \cdot \ldots \cdot p_m^{a_m}.$$

The algorithm for finding the rth symbol then reduces to computation

of values of the function

$$q = \varphi(n),$$

where q may assume values only from the set $\{a_1, a_2, \ldots, a_m\}$.

3. An nth-degree equation

$$x^n + b_1 x^{n-1} + b_2 x^{n-2} + \ldots + b_n = 0$$

(where b_i are general symbols, not specific coefficients) can be assigned a number n; it is obvious that, knowing n, one can easily reconstruct the original equation.

When $n = 2$, the equation is

$$x^2 + b_1 x + b_2 = 0.$$

Its solution may be expressed in terms of coefficients b:

$$x = -\frac{b_1}{2} \pm \sqrt[2]{\left(\frac{b_1}{2}\right)^2 - b_2}. \qquad (12.2)$$

Let us rewrite Eq. (12.2) on one line

$$x = -b_1 : 2 + -\sqrt[2]{(b_1 \times b_1 : 4 - b_2)},$$

where the square root sign applies to the entire expression in parentheses. Assume that we intend to find an expression for the solution of the nth-degree equation in terms of the radical signs. It is obvious that, whatever the form of the solution, it may consist only of the following symbols:

$$+,\ -,\ \times,\ :,\ (,\ ),\ 1,\ b_1,\ b_2,\ \ldots,\ b_n,\ \sqrt[2]{},\ \sqrt[3]{},\ \ldots,\ \sqrt[r]{}.$$

Also, we can use symbol 1 and the addition sign + to express any number which may be present as the sum $1 + 1 + 1 + \ldots + 1$. Let us code the above symbols by means of the following numerals:

$\sqrt[r]{}$	is assigned the number		$2r$	)	is assigned the number		13
+	»	»	3	1	»	»	15
−	»	»	5	b_1	»	»	17
×	»	»	7	b_2	»	»	19
:	»	»	9	. . . . .			
(	»	»	11	. . . . .			
				b_n	»	»	$(2n+15)$.

Then each expression consisting of these symbols is uniquely represented by a set of numerals. For example, the set

$$6, 11, 17, 3, 19, 13$$

corresponds to the expression

$$\overset{3}{\vee}(b_1 + b_2).$$

This set of numerals, as we already know, describes a Gödel number equal to

$$2^6 \cdot 3^{11} \cdot 5^{17} \cdot 7^3 \cdot 11^{19} \cdot 13^{13}.$$

Conversely, given a Gödel number, we can always reconstitute the corresponding set of numerals; each numeral can then be replaced by the symbol for which it stands. Thus, Gödel numbering permits us to code in a unique fashion any formula, any expression, whether it be composed of numbers or letters, signs denoting operations, or any combination of those above.

4. Assume we want to number all possible words which can be written in some alphabet A. This is easily done by matching a numeral with each character of the alphabet. Then each word will become a sequence of numerals, and we can obtain the Gödel number corresponding to each such word. And, if desired, we can also number all the sequences of such words (for example, all the deductive chains) to obtain Gödel numbers for the sequence of Gödel numbers of the individual words, as well as a Gödel number for the entire collection of such sequences.

We have now seen that the gödelization procedure reduces not only arithmetical algorithms but also any normal Markov algorithm to a computation of values of some integer-valued function. Therefore, the algorithm for such a computation is the universal algorithmic form we have sought from the beginning.

In concluding, we must point out that all of the above discussion was based on the assumption that, even though the set of data for a problem which can be processed by a given algorithm may be infinitely large, it is, nevertheless, countable. Our subsequent discussion of algorithms will also assume a countable set of conditions.

12.6. ELEMENTARY AND PRIMITIVE RECURSIVE FUNCTIONS

Functions such as $y = \varphi(x_1, x_2, \ldots, x_n)$ are called *arithmetical* if both the arguments and the functions themselves may assume values

only from the set $\{0, 1, 2, \ldots\}$. From now on, we shall discuss only arithmetical functions. Logical functions (Chapter 1) are a special case of arithmetical functions.

We shall also introduce the following system of notation:

Variables will be denoted by lower-case Latin letters:

$$a,\ b,\ c,\ \ldots,\ m,\ n,\ \ldots,\ x,\ y,\ z \quad \text{or} \quad x_1,\ x_2\ , \text{etc.}$$

Functions will be denoted by lower-case Greek letters:

$$\varphi,\ \psi,\ \chi,\ \xi,\ \ldots,\ \alpha,\ \beta,\ \gamma \quad \text{or} \quad \varphi_1,\ \varphi_2,\ \varphi_3\ , \text{etc.}$$

Predictions will be denoted by Latin capitals:

$$A,\ B,\ P,\ Q,\ R,\ S\ , \text{etc.}$$

Specific numbers or constants will also be denoted by lower-case Latin letters but will carry an asterisk:

$$a^*,\ b^*,\ x^*,\ y^*, \text{etc.}$$

We shall now define a computable arithmetical function and a solvable predicate.

A function $y = \varphi(x_1, x_2, \ldots, x_n)$ *is said to be algorithmically computable (*or just *computable) if there exists an algorithm for finding the value of this function at all values of variables* $x_1,\ x_2,\ \ldots\ x_n$.

A predicate $P(x_1, x_2, \ldots, x_n)$ *defined on the set of integers is said to be algorithmically solvable (*or just *solvable) if there exists an algorithm for finding the value of this predicate at all values of variables* $x_1, x_2, \ldots, x_n$.

These definitions are intuitive and inexact, since we have not yet defined a computing algorithm. In order to refine them, we shall have to develop the class of computable functions, starting with the most elementary computable functions.

We shall call *elementary* those arithmetical functions which can be obtained from nonnegative integers and variables by means of a finite number of additions, arithmetical subtractions (by which we mean obtaining $|x - y|$), multiplications, arithmetical divisions (by which we mean deriving the integer part of the quotient $\left[\frac{a}{b}\right]$ for $b \neq 0$), as well as from constructions involving sums and products. The computability of elementary functions is undisputable since there are algorithms for all the separate operations involved in such functions, and thus the aggregate function must also be algorithmically computable.

To construct elementary functions we need only one number, namely, 1, since

$$0 = |1 - 1|, \quad 2 = 1 + 1, \quad 3 = (1 + 1) + 1, \text{ etc.}$$

Now let us see what functions are elementary.

1. All the simple functions such as

$$\varphi(x) \equiv x + 1, \quad \varphi(y) \equiv 12y, \quad \psi(a, b, c) \equiv ab + c, \quad \chi(b) \equiv b^2$$

(since $b^2 = b \cdot b$), etc., are elementary.

2. Many of the frequently employed functions of number theory are elementary. For example:

a) $\min(x, y) = \left[\frac{|(x+y) - |x-y||}{2}\right]$;

b) $\operatorname{sg}(x) = \begin{cases} 1 & \text{when } x \geqslant 1, \\ 0 & \text{when } x = 0. \end{cases}$

Function $\operatorname{sg}(x)$ may be expressed by means of function $\min(x, y)$:

$$\operatorname{sg}(x) = \min(x, 1).*$$

From $\operatorname{sg}(x)$ one can obtain $\overline{\operatorname{sg}}(x)$:

$$\overline{\operatorname{sg}}(x) = |1 - \operatorname{sg}(x)| = \begin{cases} 1 & \text{when } x = 0, \\ 0 & \text{when } x \geqslant 1. \end{cases}$$

3. The inequality $x \leqslant y$ is equivalent to the $\min(x, y) = x$ or $|\min(x, y) - x| = 0$. Then the predicate "x is smaller than or equal to y" may be written as

$$P(x, y) = \overline{\operatorname{sg}}(|\min(x, y) - x|).$$

Indeed, if $x^* \leqslant y^*$, then $P(x^*, y^*) = 1$, that is, it is a true statement; otherwise $P = 0$.

4. Later we shall use the function

$$y \dot{-} x \begin{cases} |y - x|, & \text{if } y \geqslant x, \\ 0, & \text{if } y < x. \end{cases}$$

This is also an elementary function since it can be expressed as

$$y \dot{-} x = |y - x| \overline{\operatorname{sg}}(|\min(x, y) - x|).$$

5. The residue obtained upon division of a by n

$$\operatorname{res}(a, n) = \left|a - n\left[\frac{a}{n}\right]\right|$$

is again an elementary function.

Now, is the class of all computable functions broader than that of elementary functions? In other words, is there a computable function which is not elementary? To answer, let us follow some

*See Section 9.2f for notation.

elementary functions to see at what point they cease to be elementary.

Of the simple elementary functions, the one that increases the most rapidly is the product. The product an arises from adding a to itself n times; thus, multiplication is iterated addition.

Raising to a power is, in turn, iterated multiplication:

$$a^n = a \cdot a \cdot a \cdot \ldots \cdot a = \prod_1^n a.$$

This function is still elementary since it is expressed by a product. It increases very rapidly with a and n.

A still more rapidly increasing function involves the iteration of the operation of raising to a power:

$$\psi(0, a) = a, \quad \psi(1, a) = a^a, \quad \psi(2, a) = a^{(a^a)}$$

and, in general,

$$\psi(n+1, a) = a^{\psi(n, a)}. \tag{12.3}$$

Here the increase is so rapid that it becomes impossible to "keep pace" with the increase in $\psi(n, a)$ by devising elementary functions (for proof, see [77]); to be more precise, this function, starting from some $a = a^*$, majorizes all elementary functions; that is, for any elementary function $\varphi(a)$ there is always some m^* such that the inequality

$$\varphi(a) < \psi(m^*, a)$$

will be satisfied at all $a \geqslant a^*$. But if this is the case, then it is easy to show that the function $\gamma(n) = \psi(n, n)$ is not elementary.

Indeed, if $\psi(n, n)$ were an elementary function, then we could find some m^* (it may be assumed that $m^* \geqslant 2$ because the function $\psi(n, a)$ is monotonic) such that $\psi(n, n) < \psi(m^*, n)$ for $n \geqslant 2$. This inequality would, in particular, hold when $n = m^*$ since $m^* \geqslant 2$. We then get $\psi(m^*, m^*) < \psi(m^*, m^*)$, which is impossible.

Thus, iteration of the operation of raising to a power gives a nonelementary function. But, at the same time, $\psi(n, a)$ is *a priori* known to be computable. Indeed, suppose we want to compute $\psi(n, a)$ for any $n = n^*$, $a = a^*$. For $a = a^*$, Eq. (12.3) becomes

$$\psi(n+1, a^*) = (a^*)^{\psi(n, a^*)}.$$

Let us denote $(a^*)^m = \chi(m)$; here $\chi(m)$ is an elementary single-valued computable function, whose computation algorithm merely consists

of multiplication by a^*, repeated m times. The formula

$$\psi(n+1,\ a^*)=\chi(\psi(n,\ a^*)) \tag{12.4}$$

relates the value of ψ at a point with its value at the preceding point. Now it is sufficient to specify the initial value $\psi(0,\ a^*)=a^*$ in order to obtain a computing procedure, which successively yields

$$\begin{aligned}
&\psi(1,\ a^*)=\chi(\psi(0,\ a^*))=\chi(a^*),\\
&\psi(2,\ a^*)=\chi(\psi(1,\ a^*))=\chi(\chi(a^*)),\\
&\psi(3,\ a^*)=\chi(\psi(2,\ a^*))=\chi(\chi(\chi(a^*))),\\
&\cdots\cdots\cdots\cdots\cdots\cdots
\end{aligned}$$

This process is continued until the value of $\psi(n^*,\ a^*)$ is obtained. It is obvious that this method specifies ψ at all points and does so uniquely, since the computation of its values reduces to the computation of $\chi(m)$, which is a determinate and unique function at all points.

Our previous example has shown that computable functions need not be elementary. To continue our delineation of the class of computable functions, let us examine the method of defining $\psi(n, a)$. This function was defined by induction: that is, we were given the initial value of the function, namely, $\psi(0, a)$, and were told by what allowable operations the successive values of this function are derived from their predecessors. Now we shall use this induction method for specifying all computable functions. But first we must refine and broaden this method.

Derivation by induction can, generally speaking, be used with any ordered set in which the concepts of "predecessor" and "successor" are meaningful. Let us denote by x' the *successor function* which describes the transition to the next member of the given set. We shall assume that our given set is always $\{0,\ 1,\ 2,\ \ldots\}$, so that $0'=1;\ 1'=2;\ 2'=3$, or, in general, $x'\equiv x+1$.

The generalized procedure for defining a function $\varphi(x)$ can now be precisely defined as follows:

1. Give the value of $\varphi(0)$.

2. Specify the manner of expressing $\varphi(x')$ in terms of x and $\varphi(x)$ at any x:

$$\left.\begin{aligned}
&\varphi(0)=q,\\
&\varphi(x')=\chi(x,\ \varphi(x)).
\end{aligned}\right\} \tag{12.5}$$

In a more general case, there may also occur parameters $x_2, x_3, \ldots$ $\ldots, x_n$, which remain unaltered in the induction process. Then the defining equation (12.5) is modified to

$$\left.\begin{aligned}\varphi(0, x_2, x_3, \ldots, x_n) &= \psi(x_2, x_3, \ldots, x_n),\\ \varphi(y', x_2, x_3, \ldots, x_n) &= \chi(y, \varphi(y, x_2, x_3, \ldots, x_n), x_2, x_3, \ldots, x_n).\end{aligned}\right\} \quad (12.6)$$

If ψ and χ are known and computable functions, then the scheme of Eq. (12.6) may be used to develop a computation procedure, which will give, consecutively, $\varphi(1, x_2^*, \ldots, x_n^*)$, $\varphi(2, x_2^*, \ldots, x_n^*)$, and so on. Consequently, this scheme does indeed specify a computable function.

Let us now see which arithmetical functions can be derived by induction and how broad is the class of such functions. To state the problem exactly, we must specify which function we consider initially (that is, *a priori*) known and which operations (in addition to the above-described induction procedure) are allowable in the derivation of subsequent functions.

We shall consider the following functions as initially known (fundamental) or *primitive*:

I. $\varphi(x) = x'$, the *successor function* described above, applied to the set consisting of 0 and all natural numbers. Its abbreviated notation is S.

II. $\varphi(x_1, x_2, \ldots, x_n) = q$, where $q = \text{const}$, a *constant function*. It is denoted by C_q^n.

III. $\varphi(x_1, x_2, \ldots, x_n) = x_i$, the *identity function*.* It is denoted by U_i^n.

In addition to the induction procedure [Eqs. (12.5) or (12.6)], we shall include the *substitution procedure* IV among the allowable operations.

IV. $\varphi(x_1, x_2, \ldots, x_n) = \psi(\chi_1(x_1, x_2, \ldots, x_n), \chi_2(x_1, x_2, \ldots x_n), \ldots, \chi_m(x_1, x_2, \ldots, x_n))$.

Let us now write out all the allowable operations into one column:

$$\begin{aligned}
&\text{I. } \varphi(x) = S(x) = x'.\\
&\text{II. } \varphi(x_1, x_2, \ldots, x_n) = C_q^n = q.\\
&\text{III. } \varphi(x_1, x_2, \ldots, x_n) = U_i^n = x_i.\\
&\text{IV. } \varphi(x_1, x_2, \ldots, x_n) = \psi(\chi_1(x_1, x_2, \ldots, x_n),\\
&\qquad \chi_2(x_1, x_2, \ldots, x_n), \ldots, \chi_m(x_1, x_2, \ldots, x_n)),\\
&\text{V, a. } \begin{cases} \varphi(0) = q,\\ \varphi(x') = \chi(x, \varphi(x)). \end{cases}
\end{aligned}$$

*Rather than employ an identity function, we could consider the variables themselves originally known, as we have done in defining elementary functions. The identity function is introduced here merely for the sake of uniform exposition. Again, instead of the constant function, the null-function $\phi(x_1, x_2, \ldots, x_n) \equiv 0$ could have been used as a fundamental function, since repeated applications of the successor function then gives all the constants: $1 = 0'$, $2 = 1'$, etc.

$$\text{V, b} \quad \begin{cases} \varphi(0, x_2, \ldots, x_n) = \psi(x_2, \ldots, x_n), \\ \varphi(y', x_2, \ldots, x_n) = \chi(y, \varphi(y, x_2, \ldots \\ \qquad \ldots, x_n), x_2, x_3, \ldots, x_n). \end{cases}$$

Operations I-III specify the primitive functions and assume the role of axioms, whereas IV and V act as rules of inference.

Definition. *A function $\varphi(x_1, x_2, \ldots, x_n)$ is a primitive recursive function if it can be defined by means of finite number of applications of operations I-V.*

We shall say that a function φ *depends directly* on other functions if, for any given m and n, it satisfies operation IV for some $\psi, \chi_1, \chi_2, \ldots, \chi_m$ (in this case, φ is directly dependent on $\psi, \chi_1, \chi_2, \ldots, \chi_m$) or if, for any given q it satisfies V, a or V, b for some ψ, χ (here, φ is directly dependent on ψ and χ).

Definition. *A sequence of functions $\varphi_1, \varphi_2, \ldots, \varphi_k$ such that each function of the sequence either is primitive or depends directly on the preceding functions of the sequence while the last function φ_k is φ is called a primitive recursive description of the primitive recursive function $\varphi(x_1, x_2, \ldots, x_n)$. We shall call k the depth of the primitive recursive description of the function φ.*

A primitive recursive description is simply the series of functions obtained by successive applications of operations I-V in the definition of function φ. Indeed, we start from the initial (starting) functions (which become the beginning of our train of functions) and then proceed step by step toward the function φ.

We shall now show examples of derivation of some primitive recursive functions.

1. We define the function $\varphi(x, y)$ as:

$$\varphi(0, x) = x,$$
$$\varphi(y', x) = [\varphi(y, x)]'.$$

We thus have

$$\varphi(1, x) = x' = x + 1,$$
$$\varphi(2, x) = [\varphi(1, x)]' = (x + 1)' = x + 1 + 1 = x + 2,$$
$$\varphi(3, x) = [\varphi(2, x)]' = (x + 2)' = x + 2 + 1 = x + 3,$$

or, in general,

$$\varphi(y, x) = x + y.$$

To get a primitive recursive description of this function, write out in full the operation V, b as applied to $\varphi(y, x) = x + y$:

$$\left.\begin{aligned} \varphi(0, x) &= \psi(x), \\ \varphi(y', x) &= \chi(y, \varphi(y, x), x). \end{aligned}\right\} \tag{12.7}$$

Here, $\psi(x)$ assumes the form $\psi(x) \equiv x$, that is, it is the original identity function $\psi(x) \equiv U_1^1(x)$.

The function $\chi(y, z, x) \equiv z'$ can be obtained from the initial function $U_2^3(y, z, x) \equiv z$ by means of operation IV, where the successor function $S(z) \equiv z'$ is taken as ψ. One can, therefore, write

$$\chi(y, z, x) = S[U_2^3(y, z, x)].$$

One primitive recursive description of the function χ will be the sequence U_2^3, S, χ. Adding to it the function $\psi(x) \equiv U_1^1(x)$, on which $\varphi(y, x)$ depends directly in accordance with Eq. (12.7), we get the primitive recursive description of $\varphi(y, x)$:

$$U_2^3, S, \chi, U_1^1, \varphi.$$

2. In order to define the next primitive recursive function, we shall use the fact that the sum $x + y$ has already been defined as a primitive recursive function. We set

$$\varphi(0, x) = 0,$$
$$\varphi(y', x) = \varphi(y, x) + x.$$

Then, we obtain in succession

$$\varphi(1, x) = \varphi(0, x) + x = 0 + x = x,$$
$$\varphi(2, x) = \varphi(1, x) + x = x + x = 2x,$$
$$\varphi(3, x) = \varphi(2, x) + x = 2x + x = 3x$$
$$\cdots\cdots\cdots\cdots\cdots\cdots$$

or, in general,

$$\varphi(y, x) = yx.$$

Consequently, the product is also a primitive recursive function.

3. Using the result of Example 2, let us define

$$\varphi(0, x) = 1,$$
$$\varphi(y', x) = \varphi(y, x)\, x.$$

It is easily shown that this function means raising to a power: $\varphi(y, x) = x^y$.

4. $\varphi(0) = 1$. $\varphi(x') = \varphi(x)x'$. It can be seen easily that $\varphi(x) = x!$.

5. The function "predecessor of x"

$$\mathrm{pd}(x) = \begin{cases} 0 & \text{if} \quad x = 0, \\ |x - 1| & \text{if} \quad x > 0. \end{cases}$$

is a primitive recursive function since it is defined by operation V, a

$$\mathrm{pd}(0) = 0$$
$$\mathrm{pd}(x') = x.$$

6. The previously encountered function $x \dot{-} y$ is defined as

$$x \dot{-} 0 = x,$$
$$x \dot{-} y' = \mathrm{pd}(x \dot{-} y).$$

7. The function $\min(x, y)$ can now be defined by using operation IV:

$$\min(x, y) = y \dot{-} (y \dot{-} x).$$

8. $\max(x, y) = (x + y) \dot{-} \min(x, y)$.
9. $\mathrm{sg}(x) = \min(x, 1)$.
10. $\overline{\mathrm{sg}}(x) = 1 \dot{-} x$.
11. $|x - y| = (x \dot{-} y) + (y \dot{-} x)$.
12. The remainder obtained upon division of y by x [this function is denoted by $\mathrm{res}(y, x)$] is defined as

$$\mathrm{res}(0, x) = 0,$$
$$\mathrm{res}(y', x) = (\mathrm{res}(y, x))' \cdot \mathrm{sg}|x - (\mathrm{res}(y, x))'|.$$

13. $\left[\frac{y}{x}\right]$ is defined as

$$\left[\frac{0}{x}\right] = 0,$$
$$\left[\frac{y'}{x}\right] = \left[\frac{y}{x}\right] + \overline{\mathrm{sg}}|x - (\mathrm{res}(y, x))'|.$$

14. Primitive recursion may be used to define finite sums and products such as

$$\sum_{i=0}^{y} \varphi(i, x) \quad \text{and} \quad \prod_{i=0}^{y} \varphi(i, x).$$

Indeed,

$$\left.\begin{aligned} \sum_{i=0}^{0} \varphi(i, x) &= \varphi(0, x), \\ \sum_{i=0}^{y'} \varphi(i, x) &= \sum_{i=0}^{y} \varphi(i, x) + \varphi(y', x) \end{aligned}\right\}$$
$$\left.\begin{aligned} \prod_{i=0}^{0} \varphi(i, x) &= \varphi(0, x), \\ \prod_{i=0}^{y} \varphi(i, x) &= \prod_{i=0}^{y} \varphi(i, x) \cdot \varphi(y', x) \end{aligned}\right\}.$$

Among the primitive recursive functions just defined we find the sum $x + y$, the absolute difference $|x - y|$, the product xy, the quotient $\left[\frac{y}{x}\right]$, as well as finite sums and products. Consequently, all the elementary functions discussed at the beginning of this section are

primitive recursive functions—which is the same as saying that elementary functions are a subclass of primitive recursive functions.

12.7. PREDICATES. MINIMALIZATION

In logic, predicates are introduced whenever it is necessary to represent symbolically a relationship between several objects (see Chapter 1). In general, a predicate is defined on a set (finite or infinite) of objects and may assume two values: true or false (1 or 0). However, we shall discuss only arithmetic predicates defined on the set $\{0, 1, 2, \ldots\}$.

The predicate $P(x_1, x_2, \ldots, x_n)$ depends on n variables (it is an n-place predicate). The variables appearing under the quantifier signs in front of the predicate are *bound*; the other variables are *free*. For example, the predicate $P(x, y, z, t)$ is dependent on four variables. In $(\forall x)(\exists y)P(x, y, z, t)$,* however, x and y are bound and the predicate depends on the free variables z and t. For this reason, the expression $(\forall x)(\exists y)P(x, y, z, t)$ really represents the predicate $Q(z, t)$.

$$(\forall x)(\exists y)P(x, y, z, t) \equiv Q(z, t).$$

Indeed, this notation means the following: depending on the z^* and t^* values, there may exist for all x a y such that $P(x, y, z^*, t^*)$; or this may not be the case. In the first case Q is true (or $Q = 1$), and in the second, it is false (or $Q = 0$).

Just as a function, a predicate may be specified by induction. For example, the operation

$$E(0) \quad (\text{or} \quad E(0) = 1),$$
$$E(a') = \bar{E}(a)$$

defines the predicate $E(a) \equiv$ "a is even." By analogy with the derivation of primitive recursive functions, this points to a procedure for deriving predicates and to the concept of a "primitive recursive predicate." However, we shall not follow this path. Instead, we shall show another definition of a primitive recursive predicate—that proposed by Gödel in 1931. To start with, we define a representative function of a predicate $P(x_1, x_2, \ldots, x_n)$ as a function $\varphi(x_1, x_2, \ldots, x_n)$ which vanishes at those $x_1, x_2, \ldots, x_n$ for which

*It is read as: "for all x there exists a y such that $P(x, y, z, t)$ is true." The phase "is true" is often omitted.

$P(x_1, x_2, \ldots, x_n)$ is true and only at those. Then the assertion that $P(x_1, x_2, \ldots, x_n)$ is true may be expressed

$$\varphi(x_1, x_2, \ldots, x_n) = 0.$$

Obviously, a single predicate may have several representative functions, the zeros of which coincide.

Definition. A predicate is primitive recursive if there exists a primitive recursive function representing that predicate.

Let us assume that the predicate $Q(x_1, x_2, \ldots, x_n)$ is defined by a primitive recursive predicate $P(x_1, x_2, \ldots, x_n, y)$, using a bounded universal quantifier

$$Q(x_1, x_2, \ldots, x_n) = (\forall y)_{y \leqslant z} P(x_1, x_2, \ldots, x_n, y) \tag{12.8}$$

or, more explicitly,

$$Q(x_1, x_2, \ldots, x_n) = (\forall y)[y \leqslant z \to P(x_1, x_2, \ldots, x_n, y)].$$

The predicate $Q(x_1, x_2, \ldots, x_n)$ corresponds to the statement that, given $x_1, x_2, \ldots, x_n$, the predicate $P(x_1, x_2, \ldots, x_n, y)$ is true for all $y \leqslant z$. The predicate $Q(x_1, x_2, \ldots, x_n)$ so defined is primitive recursive since its representative function $\varphi(x_1, x_2, \ldots, x_n)$ can be expressed in terms of the representative function $\varphi(x_1, x_2, \ldots, x_n, y)$ of predicate P:

$$\varphi(x_1, x_2, \ldots, x_n) = \sum_{y=0}^{z} \varphi(x_1, x_2, \ldots, x_n, y).$$

Let us note there that z may also depend on $x_1, x_2, \ldots, x_n$; if all the inferences are to remain valid, this dependence must also be primitive recursive.

An analogous conclusion may be drawn with respect to a predicate Q defined by using a bounded existential quantifier

$$Q(x_1, x_2, \ldots, x_n) = (\exists y)_{y \leqslant z} P(x_1, x_2, \ldots, x_n, y) \tag{12.9}$$

or, more explicitly,

$$Q(x_1, x_2, \ldots, x_n) = (\exists y)[y \leqslant z \,\&\, P(x_1, x_2, \ldots, x_n, y)].$$

Here the representative function of the predicate Q is given by the product

$$\varphi(x_1, x_2, \ldots, x_n) = \prod_{y=0}^{z} \varphi(x_1, x_2, \ldots, x_n, y).$$

Now let us introduce the *minimalization operator*.

Assume that we are given a primitive recursive predicate $P(x_1, x_2, \ldots, x_n, y)$ and that it is known *a priori* that the condition

$$(\forall x_1)(\forall x_2)\ldots(\forall x_n)(\exists y)_{y\leqslant z}P(x_1, x_2, \ldots, x_n, y), \tag{12.10}$$

is satisfied; that is, for any set $x_1, x_2, \ldots, x_n$ there exists at least one $y \leqslant z$ such that $P(x_1, x_2, \ldots, x_n, y)$ will be true.

Then the predicate P may be used to define a function $\psi(x_1, x_2, \ldots, x_n)$ in the following manner: given $x_1^*, x_2^*, \ldots, x_n^*$ the value of function $\psi(x_1^*, x_2^*, \ldots, x_n^*)$ is the smallest number y^* at which $P(x_1^*, x_2^*, \ldots, x_n^*, y^*)$ is true. We shall indicate this by writing

$$\psi(x_1, x_2, \ldots, x_n) = \mu y_{y\leqslant z}P(x_1, x_2, \ldots, x_n, y). \tag{12.11}$$

By virtue of Eq. (12.10), such a y exists for all $x_1, x_2, \ldots, x_n$; consequently, $\psi(x_1, x_2, \ldots, x_n)$ is defined at all points.

If we deal with the representative function $\varphi(x_1, x_2, \ldots, x_n, y)$ of the predicate $P(x_1, x_2, \ldots, x_n, y)$ rather than the predicate itself, then Eq. (12.11) becomes

$$\psi(x_1, x_2, \ldots, x_n) = \mu y_{y\leqslant z}[\varphi(x_1, x_2, \ldots, x_n, y) = 0],$$

that is, the smallest y at which $\varphi(x_1, x_2, \ldots, x_n, y)$ vanishes is taken as the value of the function $\psi(x_1, x_2, \ldots, x_n)$.

Thus, the minimalization operator is a means for deriving new functions, starting from primitive recursive ones.

We shall now show that the function $\psi(x_1, x_2, \ldots, x_n)$, defined by means of the minimalization operator, is primitive recursive. For this purpose, we shall explicitly express the $\psi(x_1, x_2, \ldots, x_n)$ in terms of the representative function $\varphi(x_1, x_2, \ldots, x_n, y)$ of the predicate P

$$\psi(x_1, x_2, \ldots, x_n) = \sum_{k=0}^{z} \operatorname{sg}\left(\prod_{y=0}^{k}\varphi(x_1, x_2, \ldots, x_n, y)\right). \tag{12.12}$$

That Eq. (12.12) does indeed express $\psi(x_1, x_2, \ldots, x_n)$ can be verified in the following manner: let us expand expression (12.12):

$$\begin{aligned}\psi(x_1, x_2, \ldots, x_n) = {} & \operatorname{sg}[\varphi(x_1, x_2, \ldots, x_n, 0)] + \\ & + \operatorname{sg}[\varphi(x_1, x_2, \ldots, x_n, 0)\cdot\varphi(x_1, x_2, \ldots, x_n, 1)] + \\ & + \operatorname{sg}[\varphi(x_1, x_2, \ldots, x_n, 0)\cdot\varphi(x_1, x_2, \ldots, x_n, 1)\cdot \\ & \qquad \cdot\varphi(x_1, x_2, \ldots, x_n, 2)] + \ldots\end{aligned}$$

All of the above summands which include terms preceding $\varphi(x_1, x_2, \ldots, x_n, y) = 0$ are equal to 1; all succeeding summands are 0. Thus, the entire operation amounts to adding 1 to itself y times; that is, the addition gives the number y.

Since $\psi(x_1, x_2, \ldots, x_n)$ is defined in terms of sums, products, and the function $\mathrm{sg}(x)$, it is a primitive recursive function.

Previously (Section 12.6), we have cited $\mathrm{res}(a, n)$ as an example of a primitive recursive function. This function also gives the number of divisors of a. Let us divide a successively by 1, 2, 3, 4, ... and count the number of times the division gives no remainder. This will give the number of divisors of a, which we denote by $\rho(a)$. The function $\rho(a)$ is primitive recursive since

$$\rho(a) = \sum_{i=1}^{a} \overline{\mathrm{sg}}(\mathrm{res}(a, i)).$$

If a is a prime number, then $\rho(a) = 2$, since a prime number is divisible only by 1 and by itself. Then

$$\overline{\mathrm{sg}}(|\rho(a) - 2|) = \begin{cases} 1, \text{if } a \text{ is a prime number,} \\ 0 \text{ in all other cases.} \end{cases}$$

It is now easy to add up the number of prime numbers which do not exceed y. Let this number be $\pi(y)$:

$$\pi(y) = \sum_{a=2}^{y} \overline{\mathrm{sg}}(|\rho(a) - 2|).$$

The addition starts at $a = 2$, since we do not consider 0 and 1 as prime numbers: the zeroth prime number will then be 2, the first will be 3, and so on:

$$p_0 = 2,\ p_1 = 3,\ p_2 = 5,\ p_3 = 7,\ \ldots$$

Now let us tabulate some values of $\pi(y)$:

$$\begin{aligned} &\pi(2) = 1, \\ &\pi(3) = 2, \\ &\pi(4) = 2, \\ &\pi(5) = 3, \\ &\pi(6) = 3, \\ &\pi(7) = 4, \\ &\pi(8) = 4, \\ &\pi(9) = 4, \\ &\pi(10) = 4, \\ &\pi(11) = 5 \quad , \text{ etc.} \end{aligned}$$

We shall now define $p_n = \varphi(n)$ as a function which, for given n, yields the nth prime number. It is known from number theory that the nth prime number does not exceed $2^{2^{n+1}}$. We can, therefore, write

$$p_n = \mu y \left[y \leqslant 2^{2^{n+1}} \,\&\, \pi(y) = n + 1 \right],$$

since the nth prime number is the smallest such that the numbers not greater than it include exactly $n + 1$ primes. For example,

$$\varphi(1) = p_1 = \mu y [y \leqslant 16 \,\&\, \pi(y) = 2] = 3,$$
$$\varphi(2) = p_2 = \mu y [y \leqslant 256 \,\&\, \pi(y) = 3] = 5.$$

The function $p_n = \varphi(n)$ is primitive recursive since it is defined by means of the minimalization operator [the primitive recursive function $z(n) = 2^{2^{n+1}}$ acts as a bound $z(n)$ in this instance], as well as the primitive recursive relationship of $\pi(y) = n + 1$.

Let us define still another function—that giving the number of times a prime p_a occurs in the decomposition of n, and let us denote the function by $p_a(n)$. Obviously, the value of $\operatorname{ex} p_a(n)$ is the largest y for which p_a^y is still a divisor of n, or, alternatively, the smallest y at which p_a^{y+1} fails to be a divisor of n. We can then write

$$\operatorname{ex} p_a(n) = \mu y [y \leqslant n \,\&\, p_a^{y+1} \text{ is not a divisor of } n]$$

or

$$\operatorname{ex} p_a(n) = \mu y [y \leqslant n \,\&\, \operatorname{res}(n,\ p_a^{y+1}) \neq 0]. \tag{12.13}$$

Inspection of Eq. (12.13) shows that $\operatorname{ex} p_a(n)$ is a primitive recursive function. Now, the reader will recall that gödelization is associated with the decomposition of a given number into prime factors and determination of the exponent with which the prime number p_a occurs in this decomposition. Consequently, *gödelization is associated only with primitive recursive functions.*

In conclusion we shall cite, without proof, two additional primitive recursive functions. Let $m_1, m_2, \ldots, m_r$ be a set of numbers whose Gödel number is a, and let $n_1, n_2, \ldots, n_s$ be a set whose Gödel number is b. We shall now form a new sequence $m_1, m_2, \ldots, m_r, n_1, n_2, \ldots, n_s$ by appending the sequence $n_1, n_2, \ldots, n_s$ to the sequence $m_1, m_2, \ldots, m_r$. We want to determine, from the known Gödel numbers a and b, the Gödel number y for the composite sequence. The function thus defined is denoted by writing $y = a \circ b$, and is primitive recursive.

Now let $m_1, m_2, \ldots, \underline{m_i, m_{i+1}, \ldots, m_j}, m_{j+1}, \ldots, m_n$, be a sequence of numbers whose Gödel number is a. We shall cut out from this sequence the segment beginning with m_i and ending with m_j (this segment is underlined in the above expression), and insert in its place another sequence whose Gödel number is b. We want to determine the Gödel number y for the new sequence. The function giving this number in terms of known a, i, j, and b is denoted by

$$y = \operatorname{subst}_a\binom{i, j}{b}$$

and is primitive recursive.

Now recall the transformation of words in associative calculus. The operation of substitution following gödelization of an associative calculus reduces to the above inclusion operation. Consequently, transformation of words in associate calculus is also associated only with primitive recursive functions.

These conclusions will be useful in the discussion of general recursive functions.

12.8. A COMPUTABLE BUT NOT PRIMITIVE RECURSIVE FUNCTION

So far, we have dealt with primitive recursive functions. The very nature of the derivation of such functions shows that all primitive recursive functions are computable. But is the converse true? Are all computable functions primitive recursive? The answer is no. We know this from the work Péter and Ackermann who, almost simultaneously and in entirely different ways, constructed examples of a computable but not primitive recursive function. Let us follow Péter's reasoning.

Péter was the first to notice that the set of primitively recursive functions is countable. Indeed, the class of primitive functions is countable (since the number of different variables x_i and constants q is countable). Consequently, the class of primitive recursive functions, derived by a single application of operations IV or V of Section 12.6, is also countable, since the set of the sets $\psi, \chi_1, \chi_2, \ldots, \chi_m$ used in operation IV is countable, as is the set of pairs ψ, χ for operation V; this must be so since these sets are formed from elements of a countable class.

Further, the set of primitively recursive functions derived by means of two applications of operations IV or V is countable, and so on. By the same reasoning, the set of primitive recursive functions is, in general, countable. In particular, the set of primitive functions of one variable is countable (because it is contained in this countable set).

Péter succeeded in *actually numbering* all the primitive recursive functions of one variable, that is, in arranging them into a sequence

$$\varphi_0(x),\ \varphi_1(x),\ \varphi_2(x),\ \varphi_3(x),\ \ldots \tag{12.14}$$

so that from the form of a function one can determine its number, while (conversely) the form of the function is given by the

corresponding number. Then it became possible to construct an example of a computable function which is not primitively recursive.

Suppose we have a function $\psi(y, x) \equiv \varphi_y(x)$; $\psi(yx)$ is countable [since from the value $y = y^*$ one can find the corresponding function $\varphi_{y^*}(x)$ and compute its value for a given $x = x^*$; this would automatically give the value of $\psi(y^*, x^*)$]. This function is not primitive recursive. Indeed, if $\psi(y, x)$ were primitive recursive, so would $\psi(x, x)$ be, which is a function of one variable. Then $\psi(x, x) + 1$, would also be primitive recursive, since the addition of 1 constitutes an allowable operation of "succession." But since the series (12.14) contains *all* the primitive recursive functions of one variable, there would exist a number y^*, such that $\psi(x, x) + 1 \equiv \varphi_{y^*}(x)$ for all x. In other words, $\psi(x, x) + 1 \equiv \psi(y^*, x)$. Since this identity must hold for all x, it holds, in particular, for $x = y^*$. But then

$$\psi(y^*, y^*) + 1 \equiv \psi(y^*, y^*),$$

which is impossible. It means that the enumerating function $\psi(y, x)$ is not primitive recursive. This function is known, however, to be computable. Consequently, the class of primitive recursive functions does not encompass all computable functions. It must be broadened to serve our purposes.

Whereas in the case of elementary functions we were limited by the fact that we were unable to construct very rapidly increasing functions by means of allowable operations, in the case of primitive recursive functions we are limited by our *form of induction*. The trouble is that we have fixed in advance the operation (V), that is, the form in which the induction must appear.

Extension of the class of primitively recursive functions was proposed by Gödel in 1934, based on a bold idea of Herbrand.

12.9. GENERAL RECURSIVE FUNCTIONS

The Herbrand-Gödel Definition

So far, we have dealt with recursive functions, where a function φ was defined in terms of several functions χ and ψ, assumed to be known *a priori*. Now let us examine two computations using only one auxiliary function χ.

Example 1. Assume we are given the system

$$\chi(0, 4) = 7, \tag{12.15}$$
$$\chi(1, 7) = 7, \tag{12.16}$$

$$\varphi(0) = 4, \tag{12.17}$$

$$\varphi(y') = \chi(y, \varphi(y)). \tag{12.18}$$

It is required to find the chain of formal inferences which yields $\varphi(2) = 7$, starting from Eqs. (12.15) to (12.18).

1. In (12.18) we set $y = 0$:

$$\varphi(1) = \chi(0, \varphi(0)). \tag{12.19}$$

2. In (12.19) we replace $\varphi(0)$ by 4, in accordance with (12.17):

$$\varphi(1) = \chi(0, 4). \tag{12.20}$$

3. We then use (12.15):

$$\varphi(1) = 7.$$

Continuing in a similar manner, we get successively:

4. $\varphi(2) = \chi(1, \varphi(1))$.
5. $\varphi(2) = \chi(1, 7)$.
6. $\varphi(2) = 7$

Example 2. Assume that function $\varphi(n, a)$ is given by

$$\varphi(0, a) = a, \tag{12.21}$$

$$\varphi(n+1, a) = \varphi(n, a) + 1 \tag{12.22}$$

and that we want to find the value of $\varphi(3, 5)$. By a formal analysis, we shall find the operations needed for computing $\varphi(3, 5)$ by means of Eqs. (12.21) and (12.22).

1. In Eq. (12.22), we set $n = 2$, $a = 5$. Then

$$\varphi(3.5) = \varphi(2.5) + 1. \tag{12.221}$$

2. Now we set $n = 1$, $a = 5$ in (12.22) and determine $\varphi(2,5)$:

$$\varphi(2.5) = \varphi(1.5) + 1. \tag{12.222}$$

3. Again

$$\varphi(1.5) = \varphi(0.5) + 1. \tag{12.223}$$

4. We set $a = 5$ in Eq. (12.21). Then

$$\varphi(0.5) = 5. \tag{12.224}$$

5. We substitute this value of φ (0,5) into (12.223) and get

$$\varphi(1.5) = 5 + 1 = 6. \tag{12.225}$$

6. Now, substituting this value of $\varphi(1.5)$ into (12.223), we get

$$\varphi(2.5) = 6 + 1 = 7. \tag{12.226}$$

7. Finally, substituting this value of φ (2.5) into (12.221), we get

$$\varphi(3.5) = 7 + 1 = 8. \tag{12.227}$$

We required only two operations to compute the answers for the above two examples. These operations were (1) replacement of symbols (variables) by numbers and (2) substitution of equivalents, whereby we used one side of an equation as a replacement for the other (see steps 5 and 6 above).

If one can, by means of these two operations, deduce another equation from a given system of equations E, then this equation is said to be *deducible in the system E.* Since deducibility is crucial to the theory of general recursive functions, we shall consider it in detail. First, we shall introduce a broad and exact definition of deducibility. We begin by defining a "term," an "equation," and an "inference."

The letters used so far to denote functions φ, χ, $\varkappa$, σ, φ_1, φ_2, φ_3, φ_4, ... shall be called the *functional signs* (the list of functional signs is infinite). The variables will again be denoted by x, y, z, t, m, n, a, b, c, x_1, x_2, x_3, ...

We shall define a "term" by induction:

1. 0 is a term.
2. Each variable is a term.
3. R' is a term if R is a term.
4. $\varphi(R_1, R_2, \ldots, R_n)$ is a term if φ is a functional sign and $R_1, R_2, \ldots, R_n$ are terms.
5. There are no other terms.

The following are examples of forms:

1. The number 3 is a term (because 0 is a term, hence $0' = 1$ is a term, hence $1' = 2$ is a term, so that $2' = 3$ is a term).
2. Any constant is a term. Again, constants shall be denoted as x^*, y^*, z^*, t^*, m^*, n^*, ...
3. φ (2) is a term. The following are also terms:
4. $\varphi(x, y)$.
5. $\varphi(x, \chi_1(8, y), \chi_2(3, 5), \chi_3(y, z))$.
6. $\chi(x', y)$.
7. $\psi(y', x, (\varphi(z))')$, etc.

The following are not terms:

1. $\varphi(\psi)$; 2. $5(x)$; 3. $y(\varphi)$; 4. $\varphi(z, \psi)$

and so on. Thus, terms are specific expressions which are composed of symbols denoting variables, constants, and functional signs by means of brackets and primes.

Now let us define equations. *An equation shall be an expression* $R = S$, *where* R *and* S *are terms*. New equations shall be deducible from a given system of equations E by means of the following operations:

1. Substitution of numbers for symbols of variables.

2. Transformation of expressions $R = S$ and $H = P$ which do not contain variables (where R, S, H, P are terms) into an expression derived from $R = S$ by one of more simultaneous substitutions of P for occurrences of H.

The reader will recall that these were the only two operations used in the two examples considered above.

Now, our scheme for deriving primitive recursive functions involved the following "properties":

a) The values of the functions were derived from equations by a method which can be formally analyzed.

b) Each definition was arrived at by mathematical induction.

We have already established above that primitive recursive functions are a restricted class because of the mode of induction which was fixed in advance. The *a priori* fixing of the inductive method is the root of the difficulty. Were we to adopt another, possibly even a broader induction method, we would still have no guarantee that the new method would not lead to a quite restricted class of recursive functions. Herbrand therefore suggested that the induction method be left open (not fixed) and that property (a) itself be used as a definition. The Herbrand-Gödel definition of the general recursive function is as follows:

A function $\varphi(x_1, x_2, \ldots, x_n)$ *is general recursive if there exists a finite system of equations* E *such that, for any set of* $x_1^*, x_2^*, \ldots, x_n^*$, *there is one and only one* y^*, *such that the equation* $\varphi(x_1^*, x_2^*, \ldots, x_n^*) = y$ *can be deduced from* E *by a finite number of applications of operations 1 and 2* (that is, replacement of variables by numbers and substitution of equivalents).

The system E is the *defining system* of equations; one also says that E *defines the function* φ *recursively*.

This definition does not require that a function be computable from its values at preceding points; it does not require that the auxiliary functions contained in the system E be computable at all points; and no induction method is fixed *a priori*. The only requirement is that the system E define a particular value of φ (with the aid of other values of φ and values of auxiliary functions) in such a

way that φ will be uniquely computable from E at all points. Uniqueness in this instance means that E does not simultaneously yield two contradictory equations.

This definition of a general recursive function is not by itself a computation procedure. The definition merely says that if a given system of equations E recursively defines a function φ, then for any $x_1^*, x_2^*, \ldots, x_n^*$ there exists a y^* such that the equation

$$\varphi(x_1^*, x_2^*, \ldots, x_n^*) = y^*$$

can be deduced from E. But how does one go about such a deduction? How does one find y^*? One obvious way is to keep on deducing equations derivable from E until a suitable equation comes up. But that may take an infinite time. The reader will recall that a poorly organized search can lead to infinitely long wandering and no result even in a finite labyrinth. Thus some organization is a necessity if equation $\varphi(x_1^*, x_2^*, \ldots, x_n^*) = y^*$ is to be deduced in a finite, though not *a priori* bounded, number of steps. We shall not dwell on the description of the techniques employed. Suffice it to say that gödelization allows us to reduce the scanning of all the possible deductions to the application of the operator minimalization. This operator also permits another method of defining recursive functions.

12.10. EXPLICIT FORM OF GENERAL RECURSIVE FUNCTIONS

In Section 12.7 we introduced the bounded smallest-number operator which places a primitive recursive predicate $P(x_1, x_2, \ldots, x_n, y)$, or a primitive recursive function $\varphi(x_1, x_2, \ldots, x_n, y)$ representing P, into correspondence with a primitive recursive function $\psi(x_1, x_2, \ldots, x_n)$:

$$\psi(x_1, x_2, \ldots, x_n) = \mu y_{y \leqslant z} P(x_1, x_2, \ldots, x_n, y) = \tag{12.23}$$
$$= \mu y_{y \leqslant z} [\varphi(x_1, x_2, \ldots, x_n, y) = 0]$$

provided

$$(\forall x_1)(\forall x_2) \ldots (\forall x_n)(\exists y)_{y \leqslant z} P(x_1, x_2, \ldots, x_n, y)$$

or

$$(\forall x_1)(\forall x_2) \ldots (\forall x_n)(\exists y)_{y \leqslant z} [\varphi(x_1, x_2, \ldots, x_n, y) = 0],$$

where z may, in general, be a primitive recursive function of $x_1, x_2, \ldots, x_n$:

$$z = z(x_1, x_2, \ldots, x_n).$$

Let us now consider a case where the operator is not bounded. Let $\varphi(x_1, x_2, \ldots, x_n, y)$ be a primitive recursive function such that

$$(\forall x_1)(\forall x_2)\ldots(\forall x_n)(\exists y)[\varphi(x_1, x_2, \ldots, x_n, y) = 0]. \qquad (12.24)$$

Here there is no upper bound for y. The only stipulation is that for all $x_1, x_2, \ldots, x_n$ there exist a y such that

$$\varphi(x_1, x_2, \ldots, x_n, y) = 0.$$

In this case the function $\psi(x_1, x_2, \ldots, x_n)$, defined by means of the minimalization operator

$$\psi(x_1, x_2, \ldots, x_n) = \mu y\,[\varphi(x_1, x_2, \ldots, x_n, y) = 0], \qquad (12.25)$$

is *a priori* computable. Indeed, to compute its values at a point $x_1^*, x_2^*, \ldots, x_n^*$, it is sufficient to compute successively $\varphi(x_1^*, x_2^*, \ldots, x_n^*, 0)$, $\varphi(x_1^*, x_2^*, \ldots, x_n^*, 1)$, $\varphi(x_1^*, x_2^*, \ldots, x_n^*, 2)$ and so on, until one obtains a y^* such that $\varphi(x_1^*, x_2^*, \ldots, x_n^*, y^*) = 0$. The value of y^* is then the value of ψ at the point under consideration.

This computation procedure must end in a finite number of steps, because Eq. (12.24) indicates the existence of a y^* at which $\varphi = 0$. Now we want to know whether the computable function $\psi(x_1, x_2, \ldots, x_n)$ defined by Eq. (12.25) subject to condition (12.24) is general recursive. It turns out that there is a system of equations E which recursively defines ψ, that is, ψ is a general recursive function. To simplify the derivation, we shall consider a function of one variable

$$\left.\begin{aligned} &\psi(x) = \mu y\,[\varphi(x, y) = 0] \\ \text{and}\quad &(\forall x)(\exists y)[\varphi(x, y) = 0]. \end{aligned}\right\} \qquad (12.26)$$

Here, the system E, which defines $\psi(x)$ recursively, is as follows:

$$\left.\begin{aligned} &1.\ \sigma(0, x, y) = y, \\ &2.\ \sigma(z+1, x, y) = \sigma[\varphi(x, y+1), x, y+1], \\ &3.\ \psi(x) = \sigma[\varphi(x, 0), x, 0]. \end{aligned}\right\} \quad (E)$$

Let us prove that E does indeed define $\psi(x)$ recursively. Suppose that x^* is a number and we want to determine $\psi(x^*)$. According to Eq. 3 of E,

$$\psi(x^*) = \sigma[\varphi(x^*, 0), x^*, 0].$$

Now there are two possibilities: either $\varphi(x^*, 0)$, vanishes or it does not. If $\varphi(x^*, 0) = 0$, we can only use the first equation of E:

$$\sigma(0, x^*, 0) = 0, \text{ that is, } \psi(x^*) = 0.$$

But then the value of ψ must also be zero in accordance with (12.26). If, however, $\psi(x^*, 0) \neq 0$, its value may be represented as $z + 1$, and we can use the second defining equation of E:

$$\sigma[\varphi(x^*, 0), x^*, 0] = \sigma[\varphi(x^*, 1), x^*, 1].$$

Here again there are two possibilities: either $\varphi(x^*, 1)$ vanishes or it does not. If $\varphi(x^*, 1) = 0$, then we can use only Eq. 1 of E:

$$\sigma[0, x^*, 1] = 1$$

and, consequently, $\psi(x^*) = 1$. In this case Eq. 1 of E does in fact yield the value of ψ indicated by Eq. (12.26). If, however, $\varphi(x^*, 1) \neq 0$, we may represent it as $z + 1$ and again use Eq. 2 of E:

$$\sigma[\varphi(x^*, 1), x^*, 1] = \sigma[\varphi(x^*, 2), x^*, 2].$$

We continue this procedure until we find a y^* such that $\varphi(x^*, y^*) = 0$. That value of y^* will be the value of $\psi(x^*)$.

Therefore, the system E does indeed recursively define the function

$$\psi(x) = \mu y[\varphi(x, y) = 0]$$

and consequently $\psi(x)$ is a general recursive function.

In the above proof we did not use the fact that the function $\varphi(x_1, x_2, \ldots, x_n, y)$ of Eq. (12.25) is primitive recursive. For this reason, the argument holds completely even if function $\varphi(x_1, x_2, \ldots, x_n, y)$ is assumed to be general recursive.

Thus, if condition (12.24) is satisfied, the minimalization operator μy permits us to derive general recursive functions from primitive recursive functions (predicates). Further work has also shown that the difference between primitive recursive and general recursive functions resides entirely in the operator μy. Thus it has been proved that *any general recursive function* $\varphi(x_1, x_2, \ldots, x_n)$ *may be represented as*

$$\varphi(x_1, x_2, \ldots, x_n) = \psi\{\mu y[\tau(x_1, x_2, \ldots, x_n, y) = 0]\}, \qquad (12.27)$$

where ψ and τ are primitive recursive functions, while the following statement holds for the function τ:

$$(\forall x_1)(\forall x_2) \ldots (\forall x_n)(\exists y)[\tau(x_1, x_2, \ldots, x_n, y) = 0].$$

Equation (12.27) is the explicit form of general recursive functions. Let us sketch out the proof of the above statement. Assume $E = \{e_0, e_1, \ldots, e_s\}$ is a system of equations defining a function $\varphi(x_1, x_2, \ldots, x_n)$ recursively. Each equation has a Gödel number m_i.

Then the Gödel number for the entire system E is

$$w = p_0^{m_0} p_1^{m_1} p_2^{m_2} \dots p_s^{m_s}.$$

Now, we shall deduce new equations from the system E. This means that we shall successively obtain equations

$$\tilde{e}_0, \ \tilde{e}_1, \ \tilde{e}_2, \ \dots, \ \tilde{e}_r, \ \dots. \tag{12.28}$$

If n_i is the Gödel number of equation $\tilde{e}_i$, then each inference [that is, each string such as (12.28)] can be put into correspondence with the Gödel number of this inference

$$z = p_0^{n_0} p_1^{n_1} \dots p_r^{n_r}.$$

Suppose we want to evaluate φ at point $x_1^*, x_2^*, \dots, x_n^*$, that is, we wish to derive from the system E an equation of the form

$$\varphi(x_1^*, x_2^*, \dots, x_n^*) = y^*. \tag{12.29}$$

What are the properties of the Gödel number z of this inference?

1) Equations can be inferred from other equations, as we already know, by substitution of numbers for variables and replacement of occurrences. In these procedures, the Gödel numbers of the resulting new equations are primitive recursive functions of the Gödel numbers of the starting equations, since the operations of replacement of occurrences, substitution, and the determination of the Gödel number are associated only with primitive recursive functions. Some of these functions were already considered above [$\operatorname{ex} p_i(x)$, $a \circ b$, $\operatorname{subst}_a \binom{i, j}{b}$, $p_n = \varphi(n)$, etc.].

Therefore, the first requirement which z must satisfy is this: each of the exponents $n_0, n_1, n_2, \dots$ of the decomposition of z into primes must be either the Gödel number of one of the defining equations e_i or the value of some primitive recursive function of these (Gödel) numbers.

2) The last exponent n_r of the decomposition of z must be the Gödel number of an equation such as (12.29).

It turns out that the predicate

$$T(x_1^*, x_2^*, \dots, x_n^*, z) \equiv \left\{ \begin{array}{l} z \text{ is the Gödel number of the} \\ \text{inference of the value of} \\ \varphi(x_1^*, x_2^*, \dots, x_n^*) \end{array} \right\}$$

is a primitive recursive predicate. Consequently, its representing function $\tau(x_1^*, x_2^*, \dots, x_n^*, z)$ is also primitive recursive, and it is equal to zero for those z which are the Gödel numbers of inferences

terminating in the equation

$$\varphi(x_1^*, x_2^*, \ldots, x_n^*) = y^*$$

and only for those.

For this reason, our problem of finding the desired inference may be formulated as follows: find at least one number z^*, such that

$$\tau(x_1^*, x_2^*, \ldots, x_n^*, z^*) = 0. \tag{12.30}$$

Since an inference must exist at all points (by definition, E recursively defines φ recursively), the function τ has the property

$$(\forall x_1)(\forall x_2) \ldots (\forall x_n)(\exists z)[\tau(x_1, x_2, \ldots, x_n, z) = 0]. \tag{12.31}$$

Having found a z^* which Eq. (12.30) is satisfied, we can decode this Gödel number and get

$$y^* = \psi(z^*),$$

whereby ψ also turns out to be a primitive recursive function, since the decoding reduces to the following primitive recursive operations: determination of the last exponent n_r in the decomposition of z^* followed by decoding of the number n_r [which is the Gödel number of our Eq. (12.29)]. Moreover, $\psi(z)$ turns out to be a universal primitive recursive function, identical for all systems E (that is, for all general recursive functions φ), since the decoding of the Gödel number of the inference always proceeds in a standard way.

If, we we have established, any z for which

$$\tau(x_1^*, x_2^*, \ldots, x_n^*, z) = 0,$$

is the Gödel number of the desired inference, then

$$\mu z[\tau(x_1^*, x_2^*, \ldots, x_n^*, z) = 0]$$

is also the Gödel number of this inference. We shall, therefore, finally get

$$y = \varphi(x_1, x_2, \ldots, x_n) = \psi\{\mu z[\tau(x_1, x_2, \ldots, x_n, z) = 0]\},$$

where ψ and τ are primitive recursive functions and the condition (12.31) is satisfied for τ.

It should be pointed out that from the form of expression (12.30) immediately indicates that all general recursive functions form a countable set.* This conclusion arises from the fact that the number

*This conclusion could have been arrived at earlier, by observing that the set of different systems E recursively defining functions φ is countable, since all these systems E can be tagged by means of Gödel numbers.

of different recursive functions τ defining general recursive functions by means of the scheme (12.30) is also countable (denumerable).

However, in contrast with primitive recursive functions, the set of general recursive functions is not effectively countable (for further details, see Section 12.12), and, consequently, Péter's method does not allow us to construct an example of an enumerable function more general than the general recursive.

To conclude this section, let us write out the operations defining general recursive functions. Here, operations I - V are the already familiar schemes for defining primitively recursive functions, while operation VI is the explicit form of a general recursive function:

$$\begin{aligned}
&\text{I. } \varphi(x) = x', \\
&\text{II. } \varphi(x_1, x_2, \ldots, x_n) = q, \\
&\text{III. } \varphi(x_1, x_2, \ldots, x_n) = x_i, \\
&\text{IV. } \varphi(x_1, x_2, \ldots, x_n) = \psi(\chi_1(x_1, x_2, \ldots \\
&\quad \ldots, x_n), \chi_2(x_1, x_2, \ldots, x_n), \ldots, \chi_m(x_1, x_2, \ldots, x_n)), \\
&\text{Va. } \varphi(0) = q, \quad \varphi(y') = \chi(y, \varphi(y)), \\
&\text{Vb } \varphi(0, x_2, \ldots, x_n) = \psi(x_2, \ldots, x_n), \\
&\varphi(y', x_2, \ldots, x_n) = \chi(y, \varphi(y, x_2, \ldots, x_n), x_2, \ldots, x_n), \\
&\text{VI. } \varphi(x_1, x_2, \ldots, x_n) = \psi[\mu y [\tau(x_1, x_2, \ldots, x_n, y) = 0]],
\end{aligned}$$

whereby

$$(\forall x_1)(\forall x_2) \ldots (\forall x_n)(\exists y)[\tau(x_1, x_2, \ldots, x_n, y) = 0].$$

Now we can define a general recursive function: *A function $\varphi(x_1, x_2, \ldots, x_n)$ is said to be general recursive if it can be defined by using operations I - IV a finite number of times.*

Since operations I - V defining the primitive recursive functions are encompassed by operations I - VI defining general recursive functions, primitive recursive functions are a special case of general recursive functions; every primitive recursive function is a general recursive function. However, the converse is not true.

12.11. CHURCH'S THESIS

Let us now return to our initial problem, that of defining the class of computable functions. In solving this problem, we have defined in succession, the class of elementary functions, then the class of primitive recursive functions, and finally the broad class of general recursive functions. Now we must ask: is this the final solution? Or must the class of general recursive functions be further broadened?

The many attempts at broadening the class of general recursive functions have all ended in failure. And in 1936 Church suggested that *every effectively countable* function (or effectively solvable predicate) is general recursive (see [110]). By virtue of this thesis, *the class of computable functions coincides with the class of general recursive functions.*

Church's thesis cannot be proved, since it contains, on the one hand, the vague concept of a computable function and, on the other, the mathematically exact concept of a general recursive function. The thesis is a hypothesis supoorted by several valid arguments which no one has so far succeeded in refuting. One such argument is that the various refinements of the concept of an algorithm turn out to be equivalent. Thus, for instance, Markov's normal algorithm proved to be reducible to general recursive functions.

Previously we said that an "algorithm" and a "computation of the values of an arithmetical function" are identical concepts. In the light of Church's thesis a problem is algorithmically solvable only if the arithmetical function to the computation of which we reduce our problem is general recursive.

To sum up, an algorithm can exist only if a corresponding general recursive function can be constructed.

Conversely, by virtue of Church's thesis, the algorithmic unsolvability of a problem means that the arithmetical function to the computation of which the problem is reduced is not general recursive.

The proof of algorithmic unsolvability is often as involved, difficult and time-consuming as the search for an algorithm. However, algorithmic unsolvability can be proved in some cases. We shall give, without proof, two examples of this type:

Example 1. If we had an algorithm which, given the Gödel number w of an equation system E, would be capable of deciding by inspection of w whether E defines a general recursive function, then we could define once and for all which systems E define general recursive functions, and we could effectively number all such functions. In other words, we need a general recursive function $\psi(w)$ such that:

$$\psi(w) \begin{cases} = 0 \text{ if } w \text{ is the Gödel number of system } E \text{ defining a general recursive function,} \\ > 0 \text{ in all other cases.} \end{cases}$$

It has been proved [42] that such a general recursive function $\psi(w)$ does not exist. Therefore, the problem of recognizing those

systems E which define general recursive functions is algorithmically unsolvable. The set of general recursive functions turns out to be countable, but not effectively so.

Example 2. The following problem turns out to be algorithmically unsolvable: it is required to find an algorithm for recognizing, for any primitive recursive function $\tau(x, y)$ [or for any primitive recursive predicate $T(x, y)$] whether that function has the property

$$(\forall x)(\exists y)[\tau(x, y) = 0] \tag{12.32}$$

or in the case of a predicate, whether $(\forall x)(\exists y)\, T(x, y)]$.

Since primitive recursive functions can be effectively and distinctively labeled (numbered), the problem reduces to finding a computable function $\psi(r)$ such that:

$$\psi(r) \begin{cases} = 0 & \text{if } r \text{ is the number of a primitive recursive function having the property (12.32).} \\ > 0 & \text{in all other cases.} \end{cases}$$

This function proved not to be general recursive and, consequently, is noncomputable.

Even if condition (12.32) is somewhat weakened, the problem is still algorithmically unsolvable. Thus the following simple problem is algorithmically unsolvable: given a primitive recursive function $\varphi(x, y)$ it is required to find, for any x^*, whether the following condition holds for that x^*:

$$(\exists y)[\varphi(x^*, y) = 0].$$

Yet another algorithmically unsolvable problem is this: for any primitive recursive predicate $P(y)$, it is required to find whether it is true that

$$(\exists y)\, P(y).$$

In all cases, the proof reduces to proving that the corresponding recognition function is not general recursive.

One often proves the algorithmic unsolvability of a given problem by showing that it reduces to another problem, whose algorithmic unsolvability has already been proved. Sometimes it suffices to show that a narrower problem, which is a special case of the given problem, is algorithmically unsolvable. In Chapters 8 and 9, we used the method in proving algorithmic unsolvability of the two basic problems of the theory of finite automata and sequential machines.

12.12. RECURSIVE REAL NUMBERS

Recursive real numbers occur in *constructive* mathematics, an approach which has developed from the desire to avoid getting entrapped in logical contradictions (antinomies). In this approach, a proof is considered complete if, in addition to establishing a mathematical fact, one is able to demonstrate that the corresponding mathematical objects can also be computed.

The machinery of recursive functions plays an important role in constructive mathematics: it is in terms of these functions that the algorithms for effective construction of required objects are usually defined.

Consider a typical constructive formulation of a frequent practical problem. In analysis one often comes across the statement "for every small $\varepsilon > 0$ there exists a number N such that some quantity (which is a function of n) becomes smaller than ε for $n \geqslant N$." Now, what is the constructive variant of this statement? In order to arrive at it, we require:

1. A more precise definition of what we understand by "every small ε." To define such ε, we may assume, for example, $\varepsilon = \frac{1}{m}$, where m is a positive integer which may be as large as desired.

2. An effective method for determining N, starting from ε (that is, from m).

Therefore, an effective formulation of the above statement is: "there exists a general recursive function $\nu(m)$ such that some quantity which is a function of n becomes smaller than $\frac{1}{m}$ for $n \geqslant \nu(m)$."

This kind of formulation can be related to convergence of a sequence of rational numbers. Let us say that a sequence of rational numbers $a_0, a_1, a_2, \ldots, a_n, \ldots$ is recursive if there exist general recursive functions $\alpha(n), \beta(n)$ [where $\gamma \geqslant 1$] such that

$$a_n = \frac{\alpha(n) - \beta(n)}{\gamma(n)}.$$

We shall say that the *sequence converges recursively* (or *effectively)* if there exists a general recursive function $\nu(m)$ such that for any arbitrarily large $m > 0$,

$$|a_n - a_{n^*}| < \frac{1}{m}, \quad \text{if} \quad n, n^* \geqslant \nu(m).$$

The number r, defined by this effectively convergent sequence, is called *a recursive real number*.

It can be shown that the recursivity of r (that is, the fact that there exists a sequence of rational numbers which, in the limit, recursively approaches r) does not imply that r can be expanded

into a recursive decimal fraction. That is, the recursivity does not imply the existence of a general recursive function $\psi(n) \leqslant 9$ such that

$$r = \sum_{k=0}^{\infty} \frac{\psi(k)}{10^k}.$$

However, there exists one special sequence whose recursive convergence implies the possibility of recursive expansion of r in any number systems. This is the factorial expansion

$$r = \frac{a_1}{1!} + \frac{a_2}{2!} + \frac{a_3}{3!} + \ldots + \frac{a_n}{n!} + \ldots$$

where $a_n \leqslant n - 1$ for large n. If there exists a general recursive function $\chi(n)$ such that $a_n = \chi(n)$, this series always converges recursively and defines a recursive real number, which can be expanded into a recursive fraction in any number system.

Let us also mention here that the set of recursive real numbers is not larger than the set of general recursive functions; that is, this set is not larger than a countable set, whereas the set of all real numbers is a continuum. In this sense, only a very small fraction of all real numbers are recursive.

To summarize, a recursive real number is really a number which may be computed to any degree of accuracy by means of an algorithm. All numbers usually employed in mathematical analysis (e, π, $\sqrt{2}$, and so on) are recursive real numbers.

12.13. RECURSIVELY ENUMERABLE AND RECURSIVE SETS

There are several equivalent formulations defining recursive and recursively enumerable sets of integers. For convenience, we shall assemble these definitions into a table:

Recursively enumerable sets of numbers	Recursive sets of numbers
1A. A set is said to be recursively enumerable if it consists of values of some general recursive function if, alternatively, if there exists a general recursive function enumerating it, even if this involves repetitions. The empty set is deemed to be recursively enumerable. 2A. A set C containing at least one element is recursively enumerable if and only if the predicate "$y \epsilon C$" can be expressed in the form $(\exists x)P(x,y)$, where $P(x, y)$ is general recursive.	1B. A set C is said to be recursive if there exists an algorithm for determining whether a given number y belongs to C. 2B. A set C is said to be recursive if the predicate "$y \epsilon C$" can be expressed in the form $P(y)$, where P is general recursive. 3B. A set C is said to be recursive if both the set and its complement $\bar{C}$ are recursively enumerable. 4B. An infinite set C is recursive if and only if it can be enumerated by a general recursive function without repetition and in increasing order of its elements [that is if $\varphi(0)$, $\varphi(1)$, $\varphi(2)$, ... give the elements of C in increasing order].

Now, a few remarks regarding these definitions:

Note on 1A. It has been proved that, if a set can be enumerated be a general recursive function with repetition, it can also be enumerated without repetition (by another general recursive function).

Note on 2A. We shall show that definition 2A follows from 1A. Let C be the set of values of a general recursive function $\varphi(x)$. Then, the fact that a number y belongs to C means that there exists an x such that

$$y = \varphi(x),$$

or that

$$(\exists x)[y = \varphi(x)].$$

Equation $y = \varphi(x)$ may be regarded as a general recursive relationship of equality between two general recursive functions

$$P(x, y) \equiv [\chi_1(x, y) = \chi_2(x, y)],$$

where

$$\chi_1(x, y) \equiv y, \quad \chi_2(x, y) \equiv \varphi(x).$$

Note on 2B. Definition 2B is merely a more exact version of definition 1B.

Note on 3B. If condition 3B is satisfied, then it follows that condition 1B is also satisfied. Indeed, let C be enumerated by function $\varphi_1(x)$, and set $\bar{C}$ by the function $\varphi_2(x)$. To ascertain whether a given number belongs to C, we shall compute the parallel sequences

$$\varphi_1(0), \quad \varphi_1(1), \quad \varphi_1(2), \quad \varphi_1(3), \ldots \tag{I}$$

$$\varphi_2(0), \quad \varphi_2(1), \quad \varphi_2(2), \quad \varphi_2(3), \ldots \tag{II}$$

Since y belongs either to C or to $\bar{C}$, sooner or later it will appear either in row I or in row II. If it appears in row I, then $y \in C$, and if in row II, then $y \in \bar{C}$. Thus, there exists an algorithm for determining whether any y belongs to C.

Note on 4B. When condition 4B holds, there also exists an algorithm for recognizing the membership of any y in set C. Indeed, let us compute the sequence $\varphi(0), \varphi(1), \varphi(2), \ldots$. If, at some n, we arrive at $\varphi(n) > y$, then there is no need to continue the computation, and we may conclude that $y \in \bar{C}$; if, however, we find $m \leq n$ such that $\varphi(m) = y$, then $y \in C$.

We shall give a few examples of recursive sets:

1) The two—element set $\{0, 1\}$ is recursive by virtue of condition 1B or 2B.

2) Any finite set is recursive by virtue of condition 1B or 2B.

3) The set of even numbers $\{0, 2, 4, 6, 8, \ldots\}$ is recursive. Here $y \in C = [\text{res}(y, 2) = 0]$, and the set is recursive by virtue of 2B; or $\varphi(x) = 2x$, and the set is recursive by virtue of 4B.

Now, let us give examples of sets that are and sets that are not recursively enumerable.

According to definition 2A, the set of all y for which $(\exists x) P(x, y)$ at some general recursive $P(x, y)$ is recursively enumerable. One can so select $P(x, y)$ that the set $\{y\}$ will be recursively enumerable, but not recursive; its complement $\{\bar{y}\}$ will be the set of those y for which

$$\overline{(\exists x) P(x, y)} = (\forall x) \bar{P}(x, y) = (\forall x) Q(x, y)$$

(where Q is a general recursive predicate); this set $\{\bar{y}\}$ will be neither recursive nor recursively enumerable.

The set of Gödel numbers z of systems E which define a general recursive function is neither recursive nor recursively enumerable.

In conclusion, let us point out that the comparison of 1A and 4B, as well as the fact that any finite set is both recursive and recursively enumerable imply that any recursive set is recursively enumerable. The converse, however, is not true.

The concept of recursive real numbers and recursively enumerable sets is important in determining whether a machine "can do" more than just realize a given algorithm. For we have shown above that any algorithm reduces to the computation of the values of a computable integer function. *Thus, if a device generates an output of a set of numbers and that set is not recursively enumerable, we immediately know that the operation of this device cannot be represented by an algorithm; that is, this device "does more" than just realize an algorithm.*

13

Turing Machines

13.1. DESCRIPTION AND EXAMPLES OF TURING MACHINES

In Chapter 12 we showed that the fundamental, intuitively obvious requirements to which any algorithm must conform are those of determinacy, generality, and applicability (efficacy). In addition, the result of an algorithmic procedure must be completely independent of the person executing it. The executor merely acts like a machine: there is no "creative" work involved here, because the executor needs only to follow instructions. If this is so, then why not delegate the execution of the algorithm to a machine? This chapter will present one class of machines capable of executing such tasks.

The above properties of an algorithm also pertain to a machine executing this algorithm. To begin with, such a machine must be fully determinate, operating within the specified rules. Second, it must allow the input of a variety of "initial data," that is, of a variety of individual problems from a given class of problems. Third, the specified operational rules for the machine and the class of problems which can be solved must be matched in such a way that the result of machine operation will always be "readable" (that is, the machine will give a useful result).

There are many constants capable of executing algorithms. The most graphic of these is the scheme proposed in 1936 by the English mathematician Turing. We shall now describe one of the possible variants of this machine.

The basic component of our Turing machine is an infinitely long tape divided lengthwise into squares. The tape extends in only one direction (to the right), so that we can meaningfully talk about a "leftmost" square. Each square may contain only one symbol s_i from a finite alphabet $\{s_0, \ldots, s_n\}$. We shall ascribe a special significance to the symbol s_0: its presence in a square shall denote that the square is blank. In any tape, the number of nonblank squares is always finite (but as large as desired), all the other squares being blank.

The second component of the Turing machine is a *read-erase-record head*. This special device can move along the tape, either to the left or to the right, one square at a time. Upon an external command, the head can erase a symbol present in the tape square that happens to face the head at a given moment, and it can print another one in its stead. The external commands causing these actions are issued by a *controller*, a device which is itself governed by the signals generated by the head (these signals indicate the presence of symbols s_i in a given tape square). The controller operates in discrete time $t = 0, 1, 2, \ldots$, and it may assume a finite number $m + 1$ of internal states $q_0, \ldots, q_m$. Its input consists of symbols of s_i read and generated by the head, while its output consists of commands to the head (these commands indicate what symbol, if any, should be printed in a given tape square, as well as the direction of motion of the head). For example, assume that at time t the head faces the lth square from the left, that this square contains the symbol s_i, and that the controller is in state q_j. The head reads the symbol s_i and generates a signal corresponding to it. In response to this, the controller generates a symbol s_k which causes the head to erase the old symbol s_i and print s_k on the tape. Then the controller produces one of the symbols R, L, S ("right," "left," "stop"), in compliance with which the head moves one square to the right or left or stays put. After this, the controller assumes a new state q_r, which is uniquely determined by the previous state q_j and the signal s_i. After the entire operation has been completed (at time $t + 1$), the lth square contains the symbol s_k, the controller is in state q_r, and the head is situated opposite either the $(l + 1)$st, the $(l - 1)$st, or the lth square (depending on whether the motion command was R, L, or S). Thus, the controller is a sequential machine with two output converters. Its inputs are symbols from the alphabet $\{s_0, \ldots, s_n\}$, received from the read-record head. Its states are symbols from the alphabet $\{q_0, \ldots, q_m\}$. Its first output is a signal commanding the head to print a symbol from the alphabet $\{s_0, \ldots, s_n\}$, whereas its second output is a signal commanding a shift of the head and belonging to the alphabet $\{R, L, S\}$. The operation of this s-machine can be specified by means of three tables—those for an automaton and for two converters. However, it is customary to combine these into one basic table. Thus the automaton Table 13.1, the first converter Table 13.2, and the second converter Table 13.3 may all be combined, in that order, into Table 13.4, which fully describes the operation of this Turing machine. Again, if the basic table of a Turing machine is given, then its operation is uniquely specified.

Table 13.1

	s_0	s_1	s_2
q_0	q_0	q_1	q_0
q_1	q_0	q_0	q_1

Table 13.2

	s_0	s_1	s_2
q_0	s_2	s_0	s_1
q_1	s_1	s_1	s_2

Table 13.3

	s_0	s_1	s_2
q_0	R	R	L
q_1	S	L	R

Table 13.4

	s_0	s_1	s_2
q_0	$q_0 s_2 \mathrm{R}$	$q_1 s_0 \mathrm{R}$	$q_0 s_1 \mathrm{L}$
q_1	$q_0 s_1 \mathrm{S}$	$q_0 s_1 \mathrm{L}$	$q_1 s_2 \mathrm{R}$

State of the controller shall denote the *rest state* of the Turing machine; that is, row q_0 of the basic table has the following properties: (1) The first symbol in every square of this row shall always be q_0 (never q_j if $j \neq 0$); (2) The second symbol of each square will be s_i, the same symbol as in the respective *column* heading (never s_k if $k \neq i$); (3) The third symbol of every square shall be the symbol S (never R or L). For an illustration of row q_0, see Tables 13.4 and 13.5.

Table 13.5

	s_0	s_1	s_2
q_0	$q_0 s_0 \mathrm{S}$	$q_0 s_1 \mathrm{S}$	$q_0 s_2 \mathrm{S}$
q_1	$q_1 s_2 \mathrm{L}$	$q_1 s_0 \mathrm{R}$	$q_0 s_1 \mathrm{S}$

Now, if the controller is at any time t in state q_0, then whatever the position of the head, and whatever the symbol in the corresponding tape square, the controlling device will remain in state q_0, the head will not move, and the tape entries will remain the same as before. To simplify the basic table, we shall therefore omit row q_0 (see Table 13.6).

For simplicity, we shall also assume that the alphabet $\{s_i\}$ consists of only two symbols; blank (that is, 0) and nonblank (that is, 1).

Table 13.6

	s_0	s_1	s_2
q_1	$q_1 s_2 \mathrm{L}$	$q_1 s_0 \mathrm{R}$	$q_0 s_1 \mathrm{S}$

Table 13.7

Machine A

	0	1
q_1	q_0 1S	q_1 1R

Now let us discuss a few simple Turing machines.

1) *Machine A* (Table 13.7). This machine operates as follows. Assume that at $t = 0$, the controller is in state q_1, and the head faces a nonblank square. The machine then "looks" for the first blank square to the right, prints the symbol 1 in it, and stops. If, however, the head faces a blank square at $t = 0$, then the machine prints 1 in that square, and stops (no motion of the head). Tables 13.8 and 13.9 illustrate two possible modes of operation of this machine. (A bar above a tape square indicates that the head faces that square

Table 13.8

Time	Tape printout
0	. . . 1 $\overline{1}$ 1 1 1 0 0 . . . (q_1)
1	. . . 1 1 $\overline{1}$ 1 1 0 0 . . . (q_1)
2	. . . 1 1 1 $\overline{1}$ 1 0 0 . . . (q_1)
3	. . . 1 1 1 1 $\overline{1}$ 0 0 . . . (q_1)
4	. . . 1 1 1 1 1 $\overline{0}$ 0 . . . (q_1)
5	. . . 1 1 1 1 1 $\overline{1}$ 0 . . . (q_0)

Table 13.9

Time	Tape printout
0	. . . 1 0 $\overline{0}$ 0 1 . . . (q_1)
1	. . . 1 0 $\overline{1}$ 0 1 . . . (q_1)

at that time, and the symbol q shows the state of the controller.) The dots represent those tape squares where the symbols are known to remain unchanged throughout the operating time considered (the head does not reach them).

Table 13.10

Machine B

	0	1
q_1	$q_1 0 L$	$q_0 0 L$

2) *Machine B* (Table 13.10). This machine can also assume only one state (aside from state q_0). If the head faces, at $t = 0$, a nonblank square, it erases the symbol 1 in it, moves one square to the left, and stops. If the head faces, at $t = 0$, a blank square, it moves to the first nonblank square on the left, erases the symbol 1 in it, and moves one additional square to the left (see Table 13.11).

Table 13.11

Time	Tape printout
0	$\ldots 0\,1\,1\,1\,0\,\overset{q_1}{\overline{0}}\ldots$
1	$\ldots 0\,1\,1\,\overset{q_1}{\overline{1}}\,0\,0\ldots$
2	$\ldots 0\,1\,\overset{q_1}{\overline{1}}\,0\,0\,0\ldots$
3	$\ldots 0\,\overset{q_0}{\overline{1}}\,0\,0\,0\,0\ldots$

3) *Machine C* (Table 13.12). At $t = 0$, this machine may face either a blank or a nonblank square. The head then moves to the right until it finds the first group of symbols 1 after a group of

Table 13.12

Machine C

	0	1
q_1	$q_2 0 R$	$q_1 1 R$
q_2	$q_2 0 R$	$q_3 1 R$
q_3	$q_0 0 L$	$q_3 1 R$

zeros, and stops at the last 1 of that group. One version of this machine is shown in Table 13.13.

Table 13.13

Time	Tape printout
0	q_1 . . . 0 1 1 0 0 1 1 0 0 . . .
1	q_1 . . . 0 1 1 0 0 1 1 0 0 . . .
2	q_1 . . . 0 1 1 0 0 1 1 0 0 . . .
3	q_2 . . . 0 1 1 0 0 1 1 0 0 . . .
4	q_2 . . . 0 1 1 0 0 1 1 0 0 . . .
5	q_3 . . . 0 1 1 0 0 1 1 0 0 . . .
6	q_3 . . . 0 1 1 0 0 1 1 0 0 . . .
7	q_0 . . . 0 1 1 0 0 1 1 0 0 . . .

In some cases the Turing machine may be incompletely specified, in that some of the squares of the basic table contain no symbols. This is permissible in those cases where one can predict that these combinations of machine states and tape symbols will never occur. Consider an example.

Table 13.14

Machine D

	0	1
q_1		$q_2$1R
q_2	$q_3$1R	$q_2$1R
q_3	$q_3$1R	$q_4$1L
q_4		$q_0$0L

4) *Machine D* (Table 13.14). This machine searches for the group of zeros between the first two groups of ones to the right of the position of the head at $t = 0$. It then replaces all but one of these zeros with ones. If the combinations q_1, 0 and q_4, 0 are avoided at $t = 0$, they will not occur in the future because state q_1 will never occur, whereas the machine will assume state q_4 only after the last symbol 1 has been printed. One variant of this machine is shown in Table 13.15.

We shall sometimes deal with machines having not one, but several rest states (q_0', q_0'', and so on). Consider a typical example.

5) *Machine E* (Table 13.16). At $t = 0$, the head of this machine always faces a nonblank square. Then, depending on whether the next square to the left contains 0 or 1, the machine assumes either state q_0' or q_0'', and stops facing the initial square. Variants of this machine are shown in Tables 13.17 and 13.18.

Table 13.15

Time	Tape printout
0	... 0 q_1 1 1 0 0 0 1 1 0 0 ...
1	... 0 1 q_2 1 0 0 0 1 1 0 0 ...
2	... 0 1 1 q_2 0 0 0 1 1 0 0 ...
3	... 0 1 1 1 q_2 0 0 1 1 0 0 ...
4	... 0 1 1 1 1 q_3 0 1 1 0 0 ...
5	... 0 1 1 1 1 1 q_3 1 1 0 0 ...
6	... 0 1 1 1 1 1 1 q_3 1 0 0 ...
7	... 0 1 1 1 1 1 1 q_4 1 1 0 0 ...
8	... 0 1 1 1 1 1 q_0 0 1 1 0 0

In concluding this section we shall present, without special explanations, a few Turing machines which we shall need at a later stage.

6) *Machine F* (Table 13.19). This machine searches for the nearest group of 1's which follow a group of zeros to the left of the position of the head at $t = 0$.

Table 13.16

Machine E

	0	1
q_1		$q_2 1 L$
q_2	$q_0' 0 R$	$q_0'' 1 R$

Table 13.17

Time	Tape printout
0	$\overset{q_1}{\ldots\ 0\ 0\ 1\ \overline{1}\ 1\ \ldots}$
1	$\overset{q_2}{\ldots\ 0\ 0\ \overline{1}\ 1\ 1\ \ldots}$
2	$\overset{q_0''}{\ldots\ 0\ 0\ 1\ \overline{1}\ 1\ \ldots}$

Table 13.18

Time	Tape printout
0	$\overset{q_1}{\ldots\ 0\ 0\ \overline{1}\ 1\ \ldots}$
1	$\overset{q_2}{\ldots\ 0\ \overline{0}\ 1\ 1\ \ldots}$
2	$\overset{q_0'}{\ldots\ 0\ 0\ \overline{1}\ 1\ \ldots}$

Table 13.19

Machine F

	0	1
q_1	$q_2 0 L$	$q_1 1 L$
q_2	$q_2 0 L$	$q_0 1 S$

Table 13.20

Machine G

	0	1
q_1	$q_0 0 S$	$q_1 0 L$

Table 13.21

Machine H

	0	1
q_1	$q_0 1 S$	$q_2 1 R$
q_2	$q_1 0 R$	$q_2 1 R$

7) *Machine G* (Table 13.20). This machine erases all the 1's (if such symbols are present) to the left of the position of the head at $t = 0$, and continues doing so until it encounters a 0.

8) *Machine H* (Table 13.21). It differs from *Machine A* only in that it prints the symbol 1 not in the first but in the second blank square on the right.

9) *Machine I* (Table 13.22). This machine starts at a nonblank square, erases the symbol 1 in it, and transfers it to the nearest blank square on the left (in other words, it shifts a group of ones one square to the left of the starting position).

10) *Machine K* (Table 13.23). It erases the symbol 1 in the square on the right of the initial one (if that square contains a 1).

Table 13.22
Machine I

	0	1
q_1		$q_2 0 L$
q_2	$q_0 1 L$	$q_2 1 L$

Table 13.23
Machine K

	0	1
q_1	$q_2 0 R$	$q_2 1 R$
q_2	$q_0 0 S$	$q_0 0 S$

11) *Machine L* (Table 13.24). It starts at a nonblank square, moves to the left, and stops at the second blank square to the left of the first group of ones.

Table 13.24
Machine L

	0	1
q_1	$q_0 0 L$	$q_1 1 L$

Table 13.25
Machine M

	0	1
q_1	$q_0' 0 S$	$q_0'' 1 S$

12) *Machine M* (Table 13.25). It can assume two rest states. Depending on whether it faces a blank or nonblank square, it assumes state q_0' or q_0''.

13.2. THE COMPOSITION OF TURING MACHINES

As we have just seen, what a Turing machine does is uniquely determined by the controller functioning in accordance with a basic table. We shall assume that the machine always starts from an initial state denoted by q_1, and that it assumes the rest state q_0 when it ceases to work. Now we can define the operations on Turing machines so that we can derive new basic tables from the given ones. Thus imagine that we have two machines T_1 and T_2, and that at $t = 0$ the collection of symbols on the tape is such that T_1 starts operating. At this point, T_1 is in state q_1^1, and the head is opposite the l_0th square. Then, at $t = t_0^1$ machine T_1 assumes the rest state q_0^1, and the head stops opposite the l_0^1th square. Now machine T_1 shuts off, and machine T_2 takes over, starting from state q_1^2, the head of T_2 at $t = t_0^1$ facing the same square l_0^1 at which T_1 ceased operating. Then, at $t = t_0^2$, T_2 shuts off and assumes the rest state q_0^2, while its head stops opposite square l_0^2. This consecutive operation of machines T_1 and T_2, is equivalent to the operation of a single Turing machine

T, the basic table of which is synthesized according to the following rule: if the controllers of T_1 and T_2 can assume k_1 and k_2 states*, respectively, then the controller of T can have $k_1 + k_2$ states, the initial and rest states of T being q_1^1 and q_0^2, respectively. The basic table of T consists of two parts, of which the top describes T_1, and the bottom T_2. The rest state q_0^1 of T_1 is the initial state q_1^2 of T_2. For example, if T_1 is machine F (Table 13.19) and T_2 is machine G (Table 13.20), then machine T (Table 13.26) will have a table with $2 + 1 = 3$ states, where $q_0 = q_0^2$ is the rest state of G. If we recode the states of T, Table 13.26 will take the form of Table 13.27.

Table 13.26

	0	1
q_1'	$q_2'0L$	$q_1'1L$
q_2'	$q_2'0L$	$q_1''1S$
q_1''	$q_0''0S$	$q_1''0L$

Table 13.27

	0	1
q_1	$q_2 0L$	$q_1 1L$
q_2	$q_2 0L$	$q_3 1S$
q_3	$q_0 0S$	$q_3 0L$

Table 13.28

	0	1
q_1	$q_2 0S$	$q_1 0L$
q_2	$q_3 0L$	$q_2 1L$
q_3	$q_3 0L$	$q_0 1S$

Machine T so obtained is the *product* of T_1 and T_2; that is, $T = T_1 \cdot T_2$. The operation of deriving a third machine from two given ones is the multiplication of machines. Thus Tables 13.26 and 13.27 are tables of machine $F \cdot G$. Multiplication of machines is obviously a noncommutative operation: $T_1 \cdot T_2 \neq T_2 \cdot T_1$ (Table 13.28 shows the product $G \cdot F$, and it obviously differs from Table 13.27). However, multiplication is associative; that is, with three machines T_1, T_2, and T_3, we have $(T_1 \cdot T_2) \cdot T_3 = T_1 \cdot (T_2 \cdot T_3)$.** Accordingly, no parentheses are used in writing the product of several machines.

The operation of *raising to a power* is defined in the usual way: the nth power of machine T is the product obtained by multiplying T by itself n times.

So far we have discussed the multiplication of machines with one rest state. If one of the machines of the product has two or more rest states (for example, if it is machine E or M of the preceding section), the multiplication is the same, but one must indicate which of the rest states of the first constituent machine shall be the initial state of the second machine. For example, if T_1 has two rest states,

*Henceforth, the number of states shall not include the rest states, of which there may be several, as in machine M above.

**From now on, we shall omit the dots in the product of machines.

we shall write the product of T_1 and T_2 as $T=T_1\begin{cases}(1) & T_2\\ (2)\end{cases}$, or as $T=T_1\begin{cases}(1),\\ (2) & T_2\end{cases}$, depending on whether the initial state of T_2 is the first or the second rest state of T_1. Machine T also has two rest states, the first of which is one of the rest states of T_1, while the second is the rest state of T_2.

Now, the meaning of an expression such as

$$T=T_1\begin{cases}(1) & T_2\,T_3,\\ (2) & T_4\end{cases}$$

is also clear here. Here, there are two independent multiplications, involving the first and the second rest states of machine T_1.

There also exists the operation of iteration of a single machine. Thus let machine T_1 have s rest states. We select its rth rest state and make it the initial state of machine T, which is then shown as

$$T=\dot{T}_1\begin{cases}(1),\\ \cdot\\ \cdot\\ \cdot\\ (\dot{r}),\\ \cdot\\ \cdot\\ \cdot\\ (s)\end{cases}$$

This machine is the result of iteration of T_1. Here, the dots above the letters indicate that the rth rest state is made the initial state of the iterating machine T_1. If T_1 has only one rest state, then iteration yields a machine with no rest states.

Henceforth, we shall use the following notation. If we perform iteration on a machine which itself is the result of multiplication and iteration of other machines, then we place corresponding number of dots above those machines whose states (rest or initial) are used in the new machine. For example, the expression

$$T=\dot{T}_1\ddot{T}_2\begin{cases}(1) & \dot{T}_3,\\ (2) & T_4\,\ddot{T}_5,\\ (3) & T_6\end{cases}$$

means that the rest state of machine T_3 is made the initial state of machine T_1 and that the rest state of T_5 is made the initial state of T_2.

Now let us synthesize a machine by means of multiplication and iteration. Let our constituent machines be C, G, L, and M, described in Section 13.1, and let us use them to synthesize a machine N by

Machine N

	0	1
q_1	$q_2$0L	$q_1$1L
q_2	$q_3$0S	$q_4$1S
q_3	$q_5$0R	$q_3$1R
q_4	$q_1$0S	$q_4$0L
q_5	$q_5$0R	$q_6$1R
q_6	$q_0$0L	$q_6$1R

means of the above rules. Our machine N

$$N = \dot{L}M \begin{cases} (1) & C, \\ (2) & \dot{G} \end{cases}$$

will have the basic Table 13.29 (obtained from Tables 13.12, 13.20, 13.23, and 13.25). At $t = 0$, machine N is in state q_1, and its head is opposite a nonblank tape square. It then proceeds to erase all the symbols 1 to the left of its initial position, and continues doing so until it encounters two consecutive blank squares. At this point, the head returns to the right and stops opposite the extreme right nonblank square in that group of 1's opposite it at start of the operation. Table 13.30 shows one variant of N, showing only those tape conditions at which the machine assumes a new state (to reduce clutter, state symbols q_i are indicated only by their ordinal numbers i).

Here the use of iteration yielded a machine N repeating the operation of erasing groups of 1's until it is given a specified command to cease.

13.3. COMPUTATION ON TURING MACHINES

We shall show that whatever the algorithm, there will always be a Turing machine capable of executing this algorithm. To formulate this statement in precise terms, we must first formalize the concept of an algorithm in some manner. Here, we shall use Church's thesis according to which every algorithm is merely a computation of a recursive function. Because of that, we must first define what we mean by "computing" of arithmetical functions on a Turing machine.

To start with, let us specify the representation of natural numbers and zero on the tape of a Turing machine. We shall use a code in which numbers are written in the natural ("unary") system of notation, so that a number n is represented by $n+1$ symbols 1 located in consecutive squares of the tape. Thus zero is represented by a single symbol 1, unity—by two symbols 1, etc.*

*We could represent n by n consecutive symbols 1, but then we would have to represent zero by a blank square. This would interfere with our scheme, in which we need blank squares both for separating numbers and for carrying out computations.

Table 13.30

Time	Tape printout
0	1 1 0 0 1 1 1 0 1 1 0 1 1 $\overline{1}^{1}$ 1 0
. . . .	. . . . .
	1 1 0 0 1 1 1 0 1 1 $\overline{0}^{1}$ 1 1 1 1 0
	1 1 0 0 1 1 1 0 1 $\overline{1}^{2}$ 0 1 1 1 1 0
	1 1 0 0 1 1 1 0 1 $\overline{1}^{4}$ 0 1 1 1 1 0
. . . .	. . . . .
	1 1 0 0 1 1 1 $\overline{0}^{4}$ 0 0 0 1 1 1 1 0
	1 1 0 0 1 1 1 $\overline{0}^{1}$ 0 0 0 1 1 1 1 0
	1 1 0 0 1 1 $\overline{1}^{2}$ 0 0 0 0 1 1 1 1 0
	1 1 0 0 1 1 $\overline{1}^{4}$ 0 0 0 0 1 1 1 1 0
. . . .	. . . . .
	1 1 0 $\overline{0}^{4}$ 0 0 0 0 0 0 0 1 1 1 1 0
	1 1 0 $\overline{0}^{1}$ 0 0 0 0 0 0 0 1 1 1 1 0
	1 1 $\overline{0}^{2}$ 0 0 0 0 0 0 0 0 1 1 1 1 0
	1 1 $\overline{0}^{3}$ 0 0 0 0 0 0 0 0 1 1 1 1 0
	1 1 0 $\overline{0}^{5}$ 0 0 0 0 0 0 0 1 1 1 1 0
. . . .	. . . . .
	1 1 0 0 0 0 0 0 0 0 0 $\overline{1}^{5}$ 1 1 1 0
	1 1 0 0 0 0 0 0 0 0 0 1 $\overline{1}^{6}$ 1 1 0
. . . .	. . . . .
	1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 $\overline{0}^{6}$
	1 1 0 0 0 0 0 0 0 0 0 1 1 1 $\overline{1}^{0}$ 0

Two numbers are said to be located next to each other if their coded expressions are separated by a single blank square. Thus Table 13.31 contains consecutively (from the left) the numbers 3, 0, and 2.

Table 13.31

0 0 1 1 1 1 0 1 0 1 1 1 0

Now let us specify the recording and readout of arguments and values of functions on the tape. Thus assume we want machine T to compute the value of the function $\varphi(x_1, \ldots, x_n)$ of n variables $x_1, \ldots, x_n$ if the values of the arguments are $x_1 = a_1, \ldots, x_n = a_n$. Recall that the leftmost tape square of our Turing machine is considered the first one. Let us leave the first two squares blank (as in Table 13.31). Then, beginning with the third square, we record (in our code) the n consecutive numbers $a_1, \ldots, a_n$, pertaining to the independent variables $x_1, x_2, \ldots, x_n$. Table 13.32 illustrates this for $n = 3$, where $x_1 = 2$, $x_2 = 1$, and $x_3 = 3$.

Table 13.32

$$0\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ \overset{q_1}{\overline{1}}\ 0$$

From now on if there is no risk of confusion, we shall use $x_1, \ldots, x_n$ to denote both the independent variables (arguments) and the specific numerical values assumed by these variables.

We shall also say that the read-record head senses the system of numbers $(x_1, \ldots, x_n)$ in the *standard position* if these numbers are consecutively recorded on the tape, and if the head is opposite the rightmost square involved in the representation of the last number (x_n). For example, see Table 13.32, where the position of the head is, as usual, denoted by a bar.

Now we can define what we mean by computation on a Turing machine. Assume we have machine T and that the following holds at $t = 0$:

a) T is in the initial state q_1;

b) a system of n numbers $(x_1, \ldots, x_n)$ is represented on the tape, and the head senses it in the standard position;

c) all squares to the right of the one opposite the head are blank.

Now T starts working. We shall say that T computes the function $x = \varphi(x_1, \ldots, x_n)$ of n variables if, regardless of what kind of numbers $x_1, \ldots, x_n$ are involved, there arrives a time when

d) T assumes the rest state q_0;

e) the tape represents $(n + 1)$ numbers $x_1, \ldots, x_n, x$ [where $x = \varphi(x_1, \ldots, x_n)$], and the head again senses the entire number system in the standard position;

f) all squares to the right of that opposite the head are blank.

If however, the machine never stops, or if it does stop but condition (e) or (f) are not fulfilled, then this machine does not compute function φ for this particular set of arguments $(x_1, \ldots, x_n)$.

For example, to illustrate a Turing machine which does compute, let $n = 3$ and $\varphi(x_1, x_2, x_3) = x_1 + x_2 + x_3$. Then, for $x_1 = 2$, $x_2 = 1$, and $x_3 = 3$, the starting tape is that of Table 13.32, while the final tape (after the machine has stopped) is that of Table 13.33.

Table 13.33

$$\begin{array}{cccccccccccccccccccccc} & & & & & & & & & & & & & & & & & & & q_0 & & \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & \overline{1} & 0 & 0 \end{array}$$

Now we shall introduce a few specialized Turing machines.

Machine P is synthesized by multiplication and iteration of machines I, M, C, K, and A of Section 13.1:

$$P = \dot{I}M \begin{cases} (1)\ \dot{C} \\ (2)\ KCA. \end{cases}$$

This machine places numbers next to each other. It does this by transposing to the left the nonblank squares in the tape representation of a given number (see Table 13.36). The basic table is shown in Fig. 13.34. However, analysis shows that the same result can be obtained with the machine of Table 13.35, which has half the number of states.

Table 13.34

Machine P

	0	1
q_1		$q_2 0 L$
q_2	$q_3 1 L$	$q_2 1 L$
q_3	$q_4 0 S$	$q_7 1 S$
q_4	$q_5 0 R$	$q_4 1 R$
q_5	$q_5 0 R$	$q_6 1 R$
q_6	$q_1 0 L$	$q_6 1 R$
q_7	$q_8 0 R$	$q_8 1 R$
q_8	$q_9 0 S$	$q_9 0 S$
q_9	$q_{10} 0 R$	$q_9 1 R$
q_{10}	$q_{10} 0 R$	$q_{11} 1 R$
q_{11}	$q_{12} 0 L$	$q_{11} 1 R$
q_{12}	$q_0 1 S$	$q_{12} 1 R$

Table 13.35

Machine P

	0	1
q_1		$q_2 0 L$
q_2	$q_3 1 L$	$q_2 1 L$
q_3	$q_4 0 R$	$q_5 1 R$
q_4	$q_1 0 L$	$q_4 1 R$
q_5		$q_6 0 R$
q_6	$q_0 1 C$	$q_6 1 R$

Since we do not care what the actual logic of the machine is as long as we obtain the desired result, we shall understand that the machine performing the function of P can be either that of Table 13.34 or of Table 13.35. One variant of P (corresponding to Table 13.35) is shown in Table 13.36.

Machine R_m is a synthesis by multiplication and iteration of machines H, F, E, D, C, B, and A of Section **13.1**:

$$R_m = H\dot{F}^m E \begin{cases} (1) & DC^m, \\ (2) & BC^m\dot{A}, \end{cases}$$

Table **13.36**

Time	Tape printout
	1
0	0 1 1 1 0 0 0 1 1 1 $\overline{1}$ 0
	2
1	0 1 1 1 0 0 0 1 1 $\overline{1}$ 0 0
. . .	. . . . .
	2
	0 1 1 1 0 0 $\overline{0}$ 1 1 1 0 0
	3
	0 1 1 1 0 $\overline{0}$ 1 1 1 1 0 0
	4
	0 1 1 1 0 0 $\overline{1}$ 1 1 1 0 0
. . .	. . . . .
	4
	0 1 1 1 0 0 1 1 1 $\overline{0}$ 0
	1
	0 1 1 1 0 0 1 1 1 $\overline{1}$ 0 0
. . .	. . . . .
	1
	0 1 1 1 0 1 1 1 $\overline{1}$ 0 0 0
	2
	0 1 1 1 0 1 1 $\overline{1}$ 0 0 0 0
. . .	. . . . .
	2
	0 1 1 1 $\overline{0}$ 1 1 1 0 0 0 0
	3
	0 1 1 $\overline{1}$ 1 1 1 1 0 0 0 0
	5
	0 1 1 1 $\overline{1}$ 1 1 1 0 0 0 0
	6
	0 1 1 1 0 $\overline{1}$ 1 1 0 0 0 0
. . .	. . . . .
	6
	0 1 1 1 0 1 1 1 $\overline{0}$ 0 0 0
	0
	0 1 1 1 0 1 1 1 $\overline{1}$ 0 0 0

Table **13.37**

Time	Tape printout	
	1	
0	0 0 1 1 $\overline{1}$ 0 0 0 0 0	
. . .	. . . . .	H
	0 0 1 1 1 0 $\overline{1}$ 0 0 0	
. . .	. . . . .	F
	0 0 1 1 $\overline{1}$ 0 1 0 0 0	
	0_2	E
	0 0 1 1 $\overline{1}$ 0 1 0 0 0	
	0 0 1 $\overline{1}$ 0 0 1 0 0 0	B
. . .	. . . . .	C
	0 0 1 1 0 0 $\overline{1}$ 0 0 0	
	0 0 1 1 0 0 1 $\overline{1}$ 0 0	A
. . .	. . . . .	
	0 0 1 0 0 0 1 1 $\overline{1}$ 0	
. . .	. . . . .	F
	0 0 $\overline{1}$ 0 0 0 1 1 1 0	
	0_1	E
	0 0 $\overline{1}$ 0 0 0 1 1 1 0	
. . .	. . . . .	D
	0 0 1 1 $\overline{1}$ 0 1 1 1 0	
. . .	. . . . .	C
	0	
	0 0 1 1 1 0 1 1 $\overline{1}$ 0	

where the superscript m indicates the power of a given machine. For example,

$$R_1 = H\dot{F}E \begin{cases} (1) & DC, \\ (2) & BC\dot{A}. \end{cases}$$

Machine R_m operates as follows: if the numbers $x_1, \ldots, x_m$ are represented on the tape in the standard position at $t = 0$, then R_m prints the first of these numbers (that is, x_1) to the right of the representation of $(x_1, \ldots, x_m)$ and stops; after this, the tape contains a system of $m + 1$ numbers $(x_1, \ldots, x_m, x_1)$ in standard position. However, if at $t = 0$ the tape contains, in standard position, the numbers $x_1, \ldots, x_n$, where $n > m$, then R_m prints the number x_{n-m+1} on the right-hand side of the representation of this system of numbers and stops; after this, the tape contains the number system $(x_1, x_2, \ldots, x_n, x_{n-m+1})$. An example of the operation of machine R_1 is shown in Table 13.37 (the letters on the right indicate which of the component machines of R_1 are responsible for a given step of the operation; the symbols 0_1 and 0_2 are the rest states of machine E).

It can be shown that the mth power R_m^m of the machine R_m operating on numbers $(x_1, \ldots, x_m)$ in standard position, copies the entire system next to and on the right-hand side of the original representation, the final result being a system of $2m$ numbers $(x_1, \ldots, x_m, x_1, \ldots, x_m)$ in standard position. The starting and the final tapes of R_m^m at $m = 3$, $x_1 = 1$, $x_2 = 0$, and $x_3 = 2$ are shown in Table 13.38.

Table 13.38

Time	Tape printout
	$\overset{1}{}$
0	$0\,0\,1\,1\,0\,1\,0\,1\,1\,\overline{1}\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0$
. . .	. . . . .
	$\overset{0}{}$
	$0\,0\,1\,1\,0\,1\,0\,1\,1\,1\,0\,1\,1\,0\,1\,0\,1\,1\,\overline{1}\,0\,0$

Machine S_m does almost the same thing as R_m^m; it copies numbers $(x_1, \ldots, x_m)$ to the right of their original tape representation but not next to it (that is, there are two blank squares, instead of one, between x_m and x_1). The machine S_m is synthesized as follows:

$$S_m = AR_m^m BF^m BC^m.$$

The starting and final tapes of S_m at $m = 3$, $x_1 = 1$, $x_2 = 0$, and $x_3 = 2$ are shown in Table 13.39.

Now we can show *that for any recursive function there exists a Turing machine capable of computing it.* Our first proof will pertain to the basic recursive functions. These are successor, the constant and the identity functions.

I. The successor function $\varphi(x) = x'$ can be computed by machine R_1A. Indeed, if a number x is in the standard position at $t = 0$, the

Table 13.39

Time	Tape printout
0	1 0 0 1 1 0 1 0 1 1 $\overline{1}$ 0 0 0 0 0 0 0 0 0 0 0 0
. . .	. . . . .
	0 0 0 1 1 0 1 0 1 1 1 0 0 1 1 0 1 0 1 1 $\overline{1}$ 0

machine will print $\varphi(x) = x + 1$ to the right of x, after which it will stop if the tape then contains the number system $(x, x + 1)$ in standard position. Thus R_1A computes $\varphi(x) = x'$ within our definition of "computation" (see above, p. 371). The operation of R_1A at $x = 3$ is shown in Table 13.40.

II. The constant function $\varphi(x_1, \ldots, x_n) = q$ can be computed by machine HA^q. If $n = 3$, $x_1 = 1$, $x_2 = 0$, $x_3 = 0$, and $q = 2$, the tape is that of Table 13.41.

III. The identity function $\varphi(x_1, \ldots, x_n) = x_i$ can be computed by machine R_{n-i+1}. Indeed, this machine prints to the right of the number system $(x_1, \ldots, x_n)$ the $(n - i + 1)$st number from the right, that is, x_i.

The remainder of the proof on the existence of Turing machines is by induction on the depth of the recursive description of the recursive function. The depth of functions I, II, and III is, by definition, zero. The application of superposition, induction, and the

Table 13.40

Time	Tape printout
0	1 0 0 1 1 1 $\overline{1}$ 0 0 0 0 0 0 0
. . .	. . . . . R_1
	0 0 1 1 1 1 0 1 1 1 $\overline{1}$ 0 0 0 A
	0 0 1 1 1 1 0 1 1 1 1 $\overline{1}$ 0

least number operator increases the maximum depth of any function subjected to these operations by 1. Since all recursive functions can be derived from the basic functions by finite repetition of these three operations, we need only to show that there exist Turing machines

Table 13.41

Time	Tape printout	
	1	
0	0 0 1 1 0 1 0 $\overline{1}$ 0 0 0 0 0	
. . .	. . . . .	H
	0 0 1 1 0 1 0 1 0 $\overline{1}$ 0 0 0	A
	0 0 1 1 0 1 0 1 0 1 $\overline{1}$ 0 0	
	0	A
	0 0 1 1 0 1 0 1 0 1 1 $\overline{1}$ 0	

capable of realizing the operations of superposition, induction and the least number operator.

For simplicity, we shall avoid details of synthesis of Turing machines realizing these operations and shall restrict ourselves to the simpler special cases. For example, instead of dealing with superposition with respect to n variables, we shall consider superposition with respect to only one such variable.* We thus obtain the following:

IV. *Superposition.* Let M_ψ and M_χ be Turing machines computing the recursive functions $\psi(x)$ and $\chi(x)$ whose depths are α and β, respectively. We desire a Turing machine M_φ computing the function $\varphi(x) = \chi(\psi(x))$ with a depth $\max(\alpha, \beta) + 1$. This will be accomplished by the machine

$$M_\varphi = S_1 M_\psi N M_\chi N P.$$

V. *Induction.* Let M_χ be a machine computing a recursive function $\chi(x)$ of depth α. We shall devise a machine M_φ computing the recursive function $\varphi(x)$ of depth $\alpha + 1$ specified by the induction scheme

$$\varphi(0) = q, \quad \varphi(x') = \chi(\varphi(x)).$$

This machine is given by

$$M_\varphi = S_1 A^{q+1} \dot{F} E \begin{cases} (1) & BCP, \\ (2) & BCM_\chi N\dot{P}. \end{cases}$$

VI. *The smallest number operator.* Let M_χ be a machine computing the function $\chi(x, y)$ of depth α. We shall devise a machine M_φ

*For a precise description of generalized Turing machines realizing these operations, see [42].

computing the function $\varphi(x) = \mu y[\chi(x, y) = 0]$ of depth $\alpha + 1$. This machine is

$$M_\varphi = S_1 G A M_\chi E \begin{cases} (1) & GFP, \\ (2) & GF\dot{A}. \end{cases}$$

Machine M_φ will stop only if x satisfies the relation $(\exists y)[\chi(x, y) = 0]$.

We shall now illustrate the computation by induction and the use of the smallest number operator so as to elucidate the operation of machines V and VI.

Table 13.42

Time	Tape printout
	1
0	0 0 1 1 $\overline{1}$ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
. . .	. . . . . S_1
	0 0 1 1 1 0 0 1 1 $\overline{1}$ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
. . .	. . . . . A^{q+}
	0 0 1 1 1 0 0 1 1 1 0 1 $\overline{1}$ 0 0 0 0 0 0 0 0 0 0 0 0 0
. . .	. . . . . FE
	0_2
	0 0 1 1 1 0 0 1 1 $\overline{1}$ 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 B
	0 0 1 1 1 0 0 1 $\overline{1}$ 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
. . .	 C
	0 0 1 1 1 0 0 1 1 0 0 1 $\overline{1}$ 0 0 0 0 0 0 0 0 0 0 0 0
. . .	. . . . . M
	0 0 1 1 1 0 0 1 1 0 0 1 1 0 1 1 1 1 $\overline{1}$ 0 0 0 0 0 0 0
. . .	. . . . . N
	0 0 1 1 1 0 0 1 1 0 0 0 0 0 1 1 1 1 $\overline{1}$ 0 0 0 0 0 0 0
. . .	. . . . . P
	0 0 1 1 1 0 0 1 1 0 1 1 1 1 $\overline{1}$ 0 0 0 0 0 0 0 0 0 0 0
. . .	. . . . . FE
	0_2
	0 0 1 1 1 0 0 1 $\overline{1}$ 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 B
	0 0 1 1 1 0 0 $\overline{1}$ 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
. . .	 C
	0 0 1 1 1 0 0 1 0 0 1 1 1 1 $\overline{1}$ 0 0 0 0 0 0 0 0 0 0
. . .	. . . . . M
	0 0 1 1 1 0 0 1 0 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 $\overline{1}$ 0
. . .	. . . . . N
	0 0 1 1 1 0 0 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 $\overline{1}$ 0
. . .	. . . . . P
	0 0 1 1 1 0 0 1 0 1 1 1 1 1 1 $\overline{1}$ 0 0 0 0 0 0 0 0 0
. . .	. . . . . FE
	0_1
	0 0 1 1 1 0 0 $\overline{1}$ 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 B
	0 0 1 1 1 0 0 $\overline{0}$ 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
. . .	 C
	0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 $\overline{1}$ 0 0 0 0 0 0 0 0 0
. . .	. . . . . P
	0
	0 0 1 1 1 0 1 1 1 1 1 1 1 $\overline{1}$ 0 0 0 0 0 0 0 0 0 0 0 0

Table 13.43

Time	Tape printout	
	1	
0	0 0 1 1 $\overline{1}$ 0 0 0 0 0 0 0 0 0 0	
. . .	. . . . .	S_1GA
	0 0 1 1 1 0 0 $\overline{1}$ 0 0 0 0 0 0 0	
. . .	. . . . .	$M_\chi E$
	0_2	
	0 0 1 1 1 0 0 1 0 1 1 1 1 $\overline{1}$ 0	
. . .	. . . . .	G
	0 0 1 1 1 0 0 1 $\overline{0}$ 0 0 0 0 0 0	F
	0 0 1 1 1 0 0 $\overline{1}$ 0 0 0 0 0 0 0	A
	0 0 1 1 1 0 0 1 $\overline{1}$ 0 0 0 0 0 0	$M_\chi E$
. . .	. . . . .	
	0_2	
	0 0 1 1 1 0 0 1 1 0 1 1 1 $\overline{1}$ 0	
. . .	. . . . .	G
	0 0 1 1 1 0 0 1 1 $\overline{0}$ 0 0 0 0 0	F
	0 0 1 1 1 0 0 1 1 $\overline{1}$ 0 0 0 0 0	A
	0 0 1 1 1 0 0 1 1 1 $\overline{1}$ 0 0 0 0	$M_\chi E$
. . .	. . . . .	
	0_2	
	0 0 1 1 1 0 0 1 1 1 0 1 1 $\overline{1}$ 0	
. . .	. . . . .	G
	0 0 1 1 1 0 0 1 1 1 $\overline{0}$ 0 0 0 0	F
	0 0 1 1 1 0 0 1 1 1 $\overline{1}$ 0 0 0 0	A
	0 0 1 1 1 0 0 1 1 1 1 $\overline{1}$ 0 0 0	$M_\chi E$
. . .	. . . . .	
	0_2	
	0 0 1 1 1 0 0 1 1 1 1 0 1 $\overline{1}$ 0	
. . .	. . . . .	G
	0 0 1 1 1 0 0 1 1 1 1 $\overline{0}$ 0 0 0	F
	0 0 1 1 1 0 0 1 1 1 1 $\overline{1}$ 0 0 0	A
	0 0 1 1 1 0 0 1 1 1 1 1 $\overline{1}$ 0 0 0	M_L
	0 0 1 1 1 0 0 1 1 1 1 1 1 0 $\overline{1}$ 0	
	0_1	E
	0 0 1 1 1 0 0 1 1 1 1 1 1 0 $\overline{1}$ 0	
	0 0 1 1 1 0 0 1 1 1 1 1 1 $\overline{0}$ 0 0	G
	0 0 1 1 1 0 0 1 1 1 1 1 $\overline{1}$ 0 0 0	F
	0	
	0 0 1 1 1 0 1 1 1 1 $\overline{1}$ 0 0 0 0	P

Computation by induction. Let

$$\chi(y) = y + 3, \quad q = 1,$$

so that

$$\varphi(0) = 1, \quad \varphi(x') = \varphi(x+1) = \varphi(x) + 3.$$

Table 13.42 shows how machine M_φ computes the value of $\varphi(2)$. (Again, 0_1 and 0_2 denote states of machine E). The basic concept underlying this procedure consists in repeated computation of

function $\chi(y)$ while keeping a check on how many times this computation has been repeated.

The result of the computation: $\varphi(2) = 7$.

Selecting the operator of the smallest number. Let $\chi(x, y) = |x^2 - y|$, so that

$$\varphi(x) = \mu y[|x^2 - y| = 0].$$

We want to compute $\varphi(2)$.

The machine operation is shown in Table 13.43: here, we determine the consecutive values of $\chi(x, 0)$, $\chi(x, 1)$, and so on, until $\chi(x, y) = 0$. The result of the computations is $\varphi(2) = 4$.

Let us also point out that all machines synthesized according to schemes I - VI must be so designed that they will never go beyond the left-hand edge of the tape.

We have thus shown that any recursive function can be computed on a Turing machine. It can also be shown that only recursive functions may be computed on Turing machines. This is proved by means of gödelization of tape representations and verification of the fact that any change in such representations can be expressed by means of recursive functions (see the proof of this in [42]).

By virtue of the equivalence of the concepts of "recursive function" and of "function computable on a Turing machine", as well as by virtue of Church's thesis, we can now define an algorithm as follows: *An algorithm is any procedure which reduces to the computation of the values of an integer-valued function on an appropriate Turing machine.*

Conclusion

We now return to the two fundamental problems posed in the Introduction, namely: (1) Finding out what a finite automaton or a sequential machine can and cannot "do," and (2) the development of techniques for syntheses of devices which are dynamical systems of this class and perform specific tasks. The answers to these problems have been gradually accumulating in the course of our presentation of the theory. We shall now endeavor to combine the solutions scattered through previous chapters into one coherent system.

1. WHAT CAN A FINITE AUTOMATON OR A SEQUENTIAL MACHINE "DO"?

That depends whether the machine in question is autonomous or not.

If the finite automaton is autonomous, then beginning with some cycle it will only generate a periodically recurring sequence of states (the corresponding s-machine can only generate a fixed sequence of outputs). If this is a one-symbol sequence, then the machine will achieve a state of equilibrium within a finite number of cycles. If this is a multisymbol sequence, then the automaton will assume, one after another, all the states corresponding to these sequence, and will continue doing so *ad infinitum*. That is all an autonomous machine can "do."

However, regardless of what this finite periodic sequence of states is, one can always synthesize an autonomous finite automaton which will start to generate this sequence as early as its second cycle. Because of that, and because a fixed cycle of successive operations is characteristic of much of modern technology, dynamical systems which within allowable idealizations may be regarded as autonomous automata are widely used. A very old example of such automata are the animated figurines which go through complex sequences of motions, for instance, writing down a text on a piece of paper, playing predetermined music on an instrument, and so on. Modern examples range from washing machines to automatic lathes, assembly lines and control systems for cyclic operations.

If the automaton is nonautonomous, that is, its input state varies from cycle to cycle, then the answer to the question of what it can or cannot "do" can be formulated in a variety of terms, for example, in the language of representation of events. Indeed, a nonautonomous finite automaton (or s-machine) merely transforms sequences of input symbols into sequences of states (or outputs). Therefore, if we ask what such a machine can or cannot do, we are merely specifying which sequences can be transformed in a given machine and which cannot. But since the number of states (or of outputs, in the case of the s-machine) is finite, our question is equivalent to the following: which specific inputs produce each of the possible states of the automaton (or, each of the possible outputs of the s-machine)? In the terminology of the theory of finite automata this question is formulated as follows: Which events can (and which cannot) be represented by each of the possible states of the automaton (or by each of the outputs of the s-machine)? The exact answer, in terms of the necessary and sufficient conditions for representability of events in the machine, is given by the theorems of Kleene. Kleene's theorems state that only regular events can be represented in a finite automaton, a regular event being the input of sequence belonging to the class of regular sets. Thus in the language of representation of events our question receives an unambiguous answer: A finite automaton can only represent regular events.

Many important input sets encountered in practice are known to be regular. For instance, the following are regular: (a) the set consisting of any finite number of input sequences of finite length; (b) the set of any periodically repeating input sequences; (c) the set of infinitely long sequences which always terminate in specified finite sequences; and so on. However, in general, if we are faced with an infinitely long set of input sequences, we do not know *a priori* whether this set is regular or not. This is because we only have techniques for *generating* regular sets (by induction), but lack effective solution for the inverse problem of finding out whether a given set is regular or not. Thus, even though the theorems of Kleene do answer the question as to what a finite automaton can do, the answer is not an effective one. Present research attempts to construct other languages in which the answer could be given more effectively. This language problem is also of cardinal importance in the initial steps of the synthesis of automata, that is, it also figures in the second of the two problems formulated above.

Our class of dynamical systems consisting of "finite automaton" and "sequential machine," can be extended by providing the machines with an infinite memory (this can be done by letting the machine

have an infinite number of states, or providing it with an infinite tape, and so on). This gives a broader class of abstract systems—the Turing machines. The answer to the question, "what can they do?" is much simpler: they can realize any *a priori* specified algorithm. Now in modern mathematics the algorithm itself is defined as a computation of values of some recursive function. And since we know so precisely and unambiguously what a Turing machine can do, we can use this machine to define the concept of the algorithm. We thus close our chain of reasoning with the statement: an algorithm is any process which can be realized in a finite automaton supplemented by an infinite memory, that is, in a Turing machine.

2. THE SYNTHESIS OF A PRACTICAL DEVICE REALIZING A FINITE AUTOMATON OR SEQUENTIAL MACHINE

If we wish to sample the input and the output of a system only at some specified discrete times (where these instants can either be specified *a priori*, or may be the result of the very operation of the system), then we have every reason to suspect that the device embodying our requirements will be a finite automaton or an s-machine. Since the object of the design is to ensure the generation of desired outputs in response to specified inputs, we could specify our device by enumerating all the possible input-output relationships. If this enumeration results in a finite list of noncontradictory sequences of a finite length,* then we can be sure that our specification can be embodied by a finite automaton or an s-machine. Furthermore given these input-output relationships, we can derive from them the basic table of the finite automaton and the table of the output converter; together, these form one of the possible s-machines realizing the specification (the algorithm for the synthesis of such tables is described in Section 8.2**).

It is much more difficult to ensure the generation of specific outputs in response to infinitely long inputs. Such cases are frequently encountered in practice, when the duration of the operation

*We assume *a priori* that either there can be no other inputs, or that if there are such inputs, then the output may be arbitrary or, finally, that any other input will be signaled by the appearance of some symbol indicating that input.

**This type of definition is frequently encountered in the design of relay circuits, where the required input-output relationships may be enumerated, for example, by means of the so-called switching tables.

of the device being designed cannot be predetermined, and when the outputs must indicate some general properties of the input sequences, as from the first beat of the operation of the machine. In such cases a basic question arises: In what way can the relationship between the infinite input and output sequences be defined, since a direct enumeration of the sequences is impracticable in this instance. No matter how this relationship has been defined, it boils down in the end to the definition of an algorithm enabling it to be established for each beat what the input and the output symbols are in that beat.

If definition of the relationship between the infinite sequences is not restricted in some way, then from the very start we again come up against the same difficulty as that discussed at the end of the preceding section, i.e., there is no effective method of establishing whether the event which is to be represented by the automaton is regular. This means that if the language in which the definition is formulated is not restricted in some way, then there is no way of even establishing whether some finite automaton or an s-machine is capable of realizing the definition. Therefore there is no point in talking of a method for the synthesis of an automaton or an s-machine realizing the definition. Once more as a central problem arises that of finding a language sufficiently broad for the definitions of an automaton or an s-machine to be expressed in it, of great importance in technology; this language is to be such that there exist recognition algorithms as to whether there is an automaton or an s-machine capable of realizing the definition, and, when the answer is in the affirmative, the algorithms are to enable the required s-machine to be constructed.

Accordingly, in the formulation of definitions of an automaton (in the case of infinite sequences) special methods are employed (or, in other words, special languages) to avoid this difficulty. One of such methods is to write down the definition directly using the description of the regular events which the automaton is to represent, rather than the description of the correspondence between the input and the output sequences. This method is described in Section 8.4, where an effective method is indicated for the construction of the basic table of the automaton and the table of the converter, which together form the s-machine representing the given events.

Another, considerably less economical (as regards the number of the states of the automaton required) method has been described in Section 7.4, in the course of the proof of Kleene's theorems.

Other languages are also known, characterized by the fact that every definition which is expressible in the language is known to be

realized by an s-machine, and the corresponding s-machine (i.e., its tables) is effectively constructible from that expression. An example is Trakhtenbrot's predicate language, which has been briefly mentioned in the present book. The use of these languages is, in essence, based on the assumption that man is capable of nonalgorithmically (creatively) solving the problem indicated above, translating the definitions from the ordinary general language in which he thinks, into a special language in which the problem of recognition of representability of events does not arise. If one was unsuccessful in expressing a definition in such a language, the question remains open as to whether this has been caused by the fact that the definition cannot be translated into that language, and therefore realized by an s-machine, or because one failed to do so "creatively."

It follows from the foregoing that the first stage of the synthesis is in some cases carried out according to standard rules, and in some other cases it, in principle, requires creative action; but, in any case, provided the definition is realizable, the result of the first stage of the synthesis is the table of the automaton and the converter table, which form one of the s-machines realizing the definition. An s-machine so constructed is not unique; generally speaking, there is a set of other s-machines fulfilling the same definition, i.e., those equivalent to the one constructed by us, or representing it. Such s-machines may differ in the number n of the symbols in the state alphabet $\{\varkappa\}$, i.e., in the number of rows in the basic table of the automaton. The smaller the number n, the simpler is subsequent construction or the scheme of the real machine. Accordingly, the next, the second stage of the synthesis is the minimization of the machine obtained, i.e., the construction of an s-machine equivalent to the one evolved in the first stage of the synthesis and, at the same time, having the least possible number of states n.

The solution of the minimization problem depends essentially on the set of sequences which may appear at the input of the automaton during its operation. The set is of course, indicated in the original definition.

The simplest case is one where the set of the input sequences is not restricted in any way, i.e., when any sequence may appear at the input of the automaton. In this case the problem of the construction of a minimal s-machine, in the sense indicated, has been fully solved, i.e., the necessary and sufficient conditions for the minimization have been found. A method realizing the construction of a minimal S-machine involves breaking down the connection matrix into certain submatrices; it has been described in Section 9.6.

Matters are rather more complicated when the set of possible input sequences is restricted in some way. Assuming that the

constraints are arbitrary, i.e., that some arbitrary algorithm is given enabling it to be established whether a sequence satisfies the given constraints, the minimization problem turns out to be unsolvable (see Section 9.2). Accordingly, there is no minimization method suitable for any constraints and one can only attempt to find the necessary and sufficient conditions of minimization for some given particular form of constraints. However, excluding a complete sorting, even in the case of the most frequently encountered forms of constraints (e.g., when the constraint consists in only sequences of a given length appearing at the input, or sequences of any length but containing no identical symbols in succession, etc.), such necessary and sufficient conditions have not so far been found. Some observations on minimization in such cases were produced in Sections 9.4 and 9.7.

We know of only two problems with constraints imposed on the input sequences which have a full solution. These are the problem of construction of a minimal s-machine in the case when it is to operate as a finite automaton and the input sequences contain no identical symbols in succession, and the problem of construction of a minimal s-machine in the case of Aufenkamp-type constraints (see Section 9.8).

So, as a result of the second stage of the synthesis, provided it proved to be realizable, a basic table of an automaton and a converter table are constructed, which together determine an s-machine fulfilling the given definition and, at the same time have the least possible number of states. In the general case this completes the formation of the basic table of the automaton and the converter table and it is possible to pass on to the third stage of the synthesis, which consists of the construction of the abstract structure of the s-machine being designed. However, there is a particular case, frequently encountered in practice, where the input sequences are restricted, and some further work is required to construct a minimal s-machine. We are referring to the case where the rhythm of the operation of the machine being designed is determined by the change of the states at the input and there are, therefore, no input sequences containing identical symbols in succession.

In this case further work in constructing the tables of the s-machine is dependent on the technical procedures used to construct the tables. More precisely, it is essential to lay down beforehand which of two possible ways is to be followed. The first way is that of applying delay elements to a beat signal fed from outside, with special devices signaling the occurrence of a beat (i.e., a change in the input state). The second way does not require the application

of any special delay elements, but utilizes the fact that real elements have a certain inherent delay in operation and permit the construction of a machine by making use of steady states.

If the first way is used, the second stage of the synthesis described above, as far as it can be carried out to the end bearing in mind the constraints imposed on the input sequences, completes the construction of the tables of the s-machine and is immediately followed by the third stage of the synthesis: the transition to an abstract structure (see further on).

If the second way is used, further processing of the tables of the automaton and the converter is necessary. This means constructing the tables of another s-machine, which operates at a faster rhythm (as determined by the delay time in the elements employed in the construction of the s-machine), and which reproduces in its steady state the s-machine being designed, operating at a 'slow' rhythm which is determined by the moments when there is a change of state at the input.

To do this a "fast" machine satisfying this condition is first constructed, and this machine is then minimized, i.e., the second stage of the synthesis is repeated (for further details see Sections 10.2 and 10.3). In the end, by this second way we also obtain the tables of a minimal s-machine and can once more pass on to the third stage of the synthesis.

At the third stage of the synthesis an abstract structure is constructed, i.e., from the tables of the s-machine obtained in the preceding stage, the logical equations of an abstract structure representing this s-machine are set up, i.e., logical functions F_i and Φ_j in equations of the form*

$$x_i^p = F_i\left(x_1^{p-1}, x_2^{p-1}, \ldots, x_n^{p-1}; u_1^{p-1}, u_2^{p-1}, \ldots, u_s^{p-1}\right), \quad i = 1, 2, \ldots, n,$$

$$z_j^p = \Phi_j\left(x_1^p, x_2^p, \ldots, x_n^p; u_1^p, u_2^p, \ldots, u_s^p\right), \quad j = 1, 2, \ldots, l.$$

Depending on the number of states in the elements at our disposal for the construction of the machine, the functions will be those of two-, three-, and generally of m-valued logic. The method of coding and the construction of these functions is given in Section 4.2.

In the case of construction of an automaton based on steady states, the coding and construction of the functions F_i and Φ_j are given in Section 5.4.

As a result of the third stage of the synthesis the problem is reduced to one which is much more familiar to the project engineer,

*The equations are written out for a machine of the P-P type.

that of realizing a system of logical relations with the technical means at his disposal. At this point, broadly speaking, the problems of abstract synthesis are no longer relevant. Consequently, there are no problems of the general theory of finite automata and sequential machines which are applicable. From this moment on, the problem belongs to the realm of technical realizations of the abstract structure which has been obtained.

The problems arising in this connection are studied in the theory of switching circuits and the theory of logical systems, in the narrow sense of these terms. The problems solvable by these theories have hardly been considered in this book, or a mere mention of them has been made in passing.

If the subsequent construction of the scheme is based on delay elements, then the number of such elements is predetermined by the number of the equations in the abstract structure, which is known to be minimal if the second stage of the synthesis has been carried out to the end. It was just this obtaining of a scheme with a minimal number of elements of delay that constituted the second stage of the synthesis. The problem of technical realization then reduces to the construction of logical converters realizing the functions F_i and Φ_j contained in the right-hand sides of the equations of the abstract structure. From one and the same set of logical elements the converters may be constructed in various ways. This too has its own minimization problems, but these in fact concern converters and not sequential machines, i.e., they relate to statics and not to dynamics, and, therefore, only a brief mention of them has been made in Section 2.6.

If the available set of logical elements does not contain a ready-made delay element, this does not exclude the possibility of constructing schemes, since the delay element itself, being the simplest automaton, can be constructed from the elements of the set, for example, using the steady states of equilibrium.

If the entire machine is constructed using steady states, i.e., without special delay elements for the beat signal fed from without, this means that a fast machine is to be constructed according to the abstract structure obtained at the end of the third stage; elements which have an inherent delay (e.g., repeaters) serve as delayers. In particular, in the common case when the schemes are assembled from relays, the delayer for the "fast" s-machine will be the intermediate relays, while the converters F_i and Φ_j are formed from chains of contacts of the input and intermediate relays.

With such a circuit construction (using steady states), there arise additional technical difficulties, in connection with the fact

that in a nonsynchronized systems the delay time of the elements is not strictly the same. This leads to the danger of relay "competition" arising, which may sometimes result in an incorrect operation of the circuit. In such cases the danger of competition is obviated by special circuits, called realizations; in these circuits either not more than one relay operates in each beat, or the feedback circuits are artificially cut off at the switching moments.

There are various methods for the construction of realizations, only one of which has been briefly described in Section 5.4, since realization problems do not relate to the general theory of finite automata and sequential machines. In the construction of circuits using delayers, with the beat signal to them from outside, there is no likelihood of competition arising and, therefore, the realization problem does not arise.

Problems*

CHAPTER 1

1. Show that the set of points on a semicircle has the cardinality of the continuum.
2. Show that the union of two countable sets is countable.
3. Show that the set of rational numbers is countable.
4. We know that an infinite subset of a countable set is countable. Use this fact to show that the set of primes is countable.
5. Use equivalent transformations to convert the following six functions to a form containing only disjunction, conjunction, and negation:

   1. $f(x_1, x_2) = (x_1 \bigtriangledown x_2) \bigtriangledown (1/x_2)$
   2. $f(x_1, x_2) = x_2/(x_1 \leftarrow x_2)$
   3. $f(x_1, x_2) = x_1 \sim (x_1 \downarrow x_2)$
   4. $f(x_1, x_2) = x_1 \bigtriangledown (x_1/x_2)$
   5. $f(x_1, x_2, x_3) = (x_1 \downarrow x_2) \leftarrow (x_3/x_1)$
   6. $f(x_1, x_2, x_3) = (x_1/x_2) \bigtriangledown (\bar{x}_3 \rightarrow x_1)$

6. Find the complete disjunctive normal form of the function

$$f(x_1, x_2) = x_1 \bigtriangledown x_2 \,\&\, (x_1 \,\&\, x_2) .$$

7. Find the complete conjunctive normal form of the function

$$f(x_1, x_2, x_3) = (x_1 \bigtriangledown x_2) \,\&\, (\bar{x}_2 \bigtriangledown x_3) \,\&\, (x_1 \bigtriangledown \bar{x}_2) \,\&\, (x_2 \bigtriangledown x_3) .$$

8. For each of the following two functions, find the complete disjunctive and conjunctive normal forms, constructing as a preliminary the characteristic table:

$$f_1(x_1, x_2) = x_1 \rightarrow \bar{x}_2$$
$$f_2(x_1, x_2) = x_1 \sim x_2$$

---

*In all problems the symbol x corresponds to the symbol x originally used in the Russian edition and throughout the text. Reader is advised to note the difference in solving the problem.

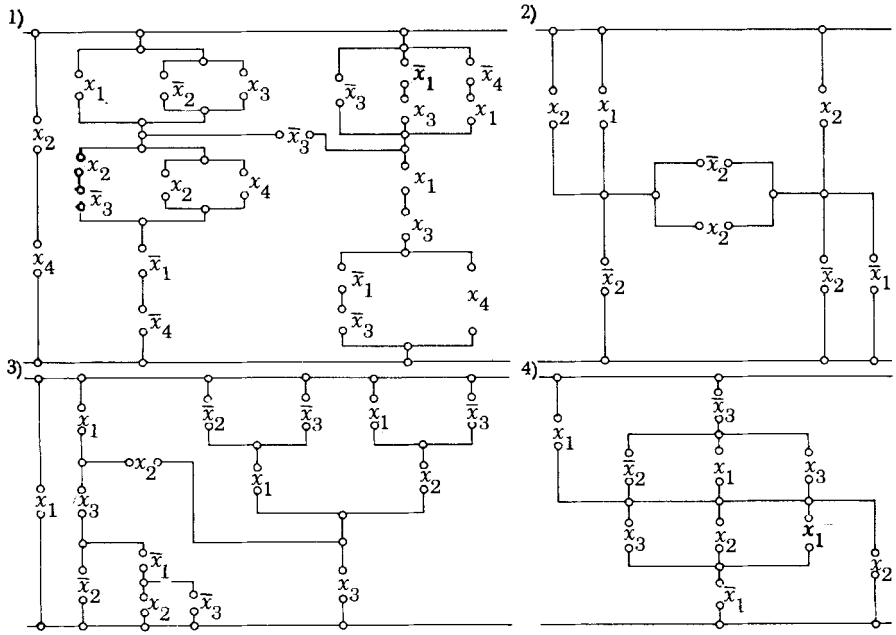
9. For the function $f(x_1, x_2) = x_1 \vee \bar{x}_2$ construct the complete disjunctive normal form, simplifying this function as a preliminary.

10. Given the predicate $P(x, y, z) = [x \to (y \to z)]$ find the predicate $Q(y, z) = (\forall x) P(x, y, z)$

11. Given the predicate $P(x, y, z) = [\overline{\kappa \vee (y \sim z)}]$ find the predicate $Q(y, z) = (\exists x) P(x, y, z)$.

CHAPTER 2

1. For the following four contact diagrams, find the simplest equivalent circuits:



2. Each of the following two tables gives the values of two logical functions y_1 and y_2 (all told, four functions). Construct the contact diagrams corresponding to these functions

   a) by the canonical method,

   b) by the block method.

Table 1

x_1	0	I	0	I
x_2	0	0	I	I
y_1	I	I	I	0
y_2	0	I	I	0

Table 2

x_1	0	I	0	I	0	I	0	I
x_2	0	0	I	I	0	0	I	I
x_3	0	0	0	0	I	I	I	I
y_1	I	0	I	0	I	0	0	I
y_2	0	I	0	0	0	I	I	0

3. From a given logical function $y = (x_1 \,\&\, x_2) \triangledown (\bar{x}_1 \,\&\, x_2)$ construct the diagram at the diodes that realize this function.

4. From a given logical function $y = (x_2 \,\&\, x_3) \,\&\, \bar{x}_1$ construct the scheme on the triodes that realize this function.

5. With the aid of Quine's algorithm, find the minimum disjunctive normal form of the following functions:

$$y(x_1, x_2, x_3) = (x_1 \,\&\, \bar{x}_2 \,\&\, x_3) \triangledown (\bar{x}_1 \,\&\, \bar{x}_2 \,\&\, x_3) \triangledown (x_1 \,\&\, x_2 \,\&\, x_3) \triangledown (x_1 \,\&\, x_2 \,\&\, \bar{x}_3)$$

$$y(x_1, x_2, x_3) = (\bar{x}_1 \,\&\, \bar{x}_2 \,\&\, \bar{x}_3) \triangledown (x_1 \,\&\, x_2 \,\&\, x_3) \triangledown (x_1 \,\&\, \bar{x}_2 \,\&\, x_3) \triangledown (x_1 \,\&\, \bar{x}_2 \,\&\, \bar{x}_3).$$

CHAPTER 3

1. Give an example of a dynamic system that can be treated as a finite automaton.

2. The table below is a combined table of an automaton and a transformer. Construct the graph and the interconnection matrix of unions of this sequential machine.

x \ ρ	ρ_1	ρ_2
x_1	$x_4\lambda_0$	$x_1\lambda_1$
x_2	$x_1\lambda_0$	$x_3\lambda_1$
x_3	$x_2\lambda_1$	$x_1\lambda_0$
x_4	$x_2\lambda_0$	$x_2\lambda_1$

3. Suppose that you are given the interconnection matrix C shown. Construct a table of an automaton of the P-P type and the table of transformations.

$$C = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{array} \begin{array}{c} \begin{array}{ccccc} x_1 & x_2 & x_3 & x_4 & x_5 \end{array} \\ \begin{bmatrix} 0 & \rho_1\lambda_1 & \rho_3\lambda_0 & 0 & \rho_2\lambda_1 \\ \rho_3\lambda_0 & \rho_1\lambda_0 & 0 & \rho_2\lambda_0 & 0 \\ 0 & \rho_2\lambda_1 & \rho_1\lambda_1 & \rho_3\lambda_1 & 0 \\ 0 & 0 & \rho_2\lambda_0 \vee \rho_3\lambda_1 & 0 & \rho_1\lambda_1 \\ \rho_1\lambda_0 \vee \rho_3\lambda_0 & 0 & 0 & \rho_2\lambda_0 & 0 \end{bmatrix} \end{array}$$

4. Suppose that we have an s-machine of the P-Pr type defined by two tables, namely, the basic table of the automaton and the table of the output transformer. Construct its diagram of states and interconnection matrix.

Automaton

x \ ρ	ρ_1	ρ_2
x_1	x_2	x_5
x_2	x_3	x_4
x_3	x_4	x_3
x_4	x_2	x_5
x_5	x_1	x_5

Transformer

x \ ρ	ρ_1	ρ_2
x_1	λ_1	–
x_2	λ_1	–
x_3	λ_0	λ_1
x_4	λ_0	λ_0
x_5	–	λ_0

5. Do the same thing for the following machine of the P-Pr type:

Automaton

x \ ρ	ρ_1	ρ_2
x_1	x_2	x_3
x_2	x_1	x_3
x_3	x_6	x_6
x_4	x_3	x_4
x_5	x_6	x_3
x_6	x_2	x_6

Transformer

x \ ρ	ρ_1	ρ_2
x_1	λ_0	–
x_2	λ_1	–
x_3	λ_0	λ_1
x_4	–	λ_0
x_5	–	–
x_6	λ_1	λ_1

6. On the basis of the examples given, state the general properties of the interconnection matrix of an s-machine of the P-Pr type.

7. Suppose that we are given a finite automaton of the P-Pr type with output transformer as shown:

Automaton

x \ ρ	ρ_1	ρ_2	ρ_3
x_1	x_4	x_2	x_5
x_2	x_3	x_3	x_5
x_3	x_5	x_1	x_2
x_4	x_1	x_2	x_2
x_5	x_3	x_3	x_2

Transformer

x	x_1	x_2	x_3	x_4	x_5
λ	λ_2	λ_3	λ_1	λ_2	λ_3

Is the entire system an automaton, that is, does there exist a single-valued function F^* such that $\lambda^p = F^*(\lambda^{p-1}, \rho^p)$.

8. Suppose that we are given an s-machine of the P-P type together with a table of the automaton and the output transformer. Let us assign to each pair $\rho_i\lambda_j$ the symbol θ_k from the alphabet $\{\theta_1, \theta_2, \theta_3, \ldots, \theta_{16}\}$. Is there a single-valued function F^* such that $\theta^p = F^*(\theta^{p-1}, \rho^p)$?

x \ ρ	ρ_1	ρ_2	ρ_3	ρ_4
x_1	$x_1\lambda_3$	$x_3\lambda_2$	$x_2\lambda_1$	$x_4\lambda_1$
x_2	$x_3\lambda_1$	$x_1\lambda_4$	$x_5\lambda_3$	$x_2\lambda_2$
x_3	$x_3\lambda_2$	$x_6\lambda_3$	$x_4\lambda_2$	$x_3\lambda_4$
x_4	$x_2\lambda_3$	$x_4\lambda_2$	$x_2\lambda_1$	$x_1\lambda_1$
x_5	$x_1\lambda_4$	$x_3\lambda_1$	$x_5\lambda_4$	$x_4\lambda_3$
x_6	$x_3\lambda_1$	$x_4\lambda_4$	$x_5\lambda_3$	$x_6\lambda_2$

CHAPTER 4

1. Construct a block diagram of the automaton described by table 4.10 in the following cases:

   (a) for every $i, k_i = 2$ and $r_i = 3$;

   (b) for every $i, k_i = 3$ and $r_i = 2$;

   (c) for every $i, k_i = r_i = 3$.

2. From the neurons of McCulloch and Pitts all logical functions of two variables.

3. Construct a trigger from the neurons of McCulloch and Pitts.

CHAPTER 5

1. From the following four tables, determine the types of automata or sequential machines.

Table 1

x \ p	p_1	p_2	p_3	p_4
x_1	$x_2\lambda_1$	$x_1\lambda_1$	$x_1\lambda_1$	$x_3\lambda_1$
x_2	$x_1\lambda_3$	$x_3\lambda_3$	$x_2\lambda_3$	$x_2\lambda_3$
x_3	$x_1\lambda_2$	$x_2\lambda_2$	$x_2\lambda_2$	$x_3\lambda_2$

Table 2

x \ p	p_1	p_2	p_3	p_4
x_1	$x_2\lambda_1$	$x_1\lambda_1$	$x_1\lambda_1$	$x_3\lambda_2$
x_2	$x_1\lambda_3$	$x_2\lambda_2$	$x_2\lambda_3$	$x_2\lambda_3$
x_3	$x_1\lambda_2$	$x_2\lambda_2$	$x_1\lambda_1$	$x_3\lambda_2$

Table 3

x \ p	p_1	p_2	p_3	p_4
x_1	$x_2\lambda_1$	$x_1\lambda_1$	$x_1\lambda_3$	$x_3\lambda_3$
x_2	$x_1\lambda_3$	$x_3\lambda_2$	$x_2\lambda_2$	$x_2\lambda_1$
x_3	$x_1\lambda_3$	$x_2\lambda_3$	$x_2\lambda_2$	$x_3\lambda_3$

Table 4

x \ p	p_1	p_2	p_3	p_4
x_1	$x_2\lambda_1$	$x_1\lambda_2$	$x_1\lambda_2$	$x_3\lambda_3$
x_2	$x_1\lambda_2$	$x_3\lambda_3$	$x_2\lambda_1$	$x_2\lambda_1$
x_3	$x_1\lambda_2$	$x_2\lambda_1$	$x_2\lambda_1$	$x_3\lambda_3$

2. From the tables of the preceding exercise, construct the tables of transitions and minimize them.

3. From the same tables, construct $(2S_0 + 1)$ realizations of Huffman.

CHAPTER 6

1. Synthesize an s-machine with input alphabet $\{\rho_1, \rho_2, \rho_3\}$ and output $\{\lambda_1\lambda_2\}$ such that, for an initial state x_0 and fixed ρ^*, if $\rho^* = \rho_1$, the periodic sequence $\lambda_1\lambda_1\lambda_2\lambda_1$ will be generated; if $\rho^* = \rho_2$, the periodic sequence $\lambda_1\lambda_2\lambda_2$ will be generated; for $\rho^* = \rho_3$, the periodic sequence $\lambda_2\lambda_1\lambda_2$ will be generated.

2. Do the same thing as in problem 1 but with the alphabet $\rho = \{\rho_1, \rho_2\}$ and the alphabet $\lambda = \{\lambda_1, \lambda_2, \lambda_3\}$
If $\rho^* = \rho_1$, the sequence $\lambda_3\lambda_1\lambda_1\lambda_2\lambda_2\lambda_1\lambda_2\lambda_2$ will be generated with period $\lambda_1\lambda_2\lambda_2$;
if $\rho^* = \rho_2$, the sequence $\lambda_2\lambda_3\lambda_2\lambda_3\lambda_1\lambda_1\lambda_1\lambda_1$ will be generated with period λ_1.

3. A periodic input sequence is applied at the input of an arbitrary s-machine. Show that the periodic sequence of output symbols is determined by a finite number of moments at the output of that machine.

CHAPTER 7

1. Show that the events mentioned in examples 1-14 of Section 7.2 are regular and, by using the concept of chains of triads, construct automata representing these events.

2. Suppose that we are given the alphabet $\{\rho_1, \rho_2\}$. The set L contains all words consisting of letters of that alphabet with the exception of words in which the same letter occurs twice in a row. Show that the set L is regular. Write the regular expression for it.

3. Do the same thing as in problem 2 for the alphabet $\{\rho_1, \rho_2, \rho_3\}$. Is the assertion of regularity of the set L so constructed true for an arbitrary finite alphabet?

4. What event is represented by the automaton shown in Fig. 3 of Chapter 2 by means of the set of events $\{x_2, x_3\}$ if it begins from the initial state x_1? Write the regular expression for this event.

5. Show that the intersection of two regular sets is regular.

6. An s-machine is said to be strongly connected if, for every pair x_i and x_j of states of that machine there exists an input sequence that takes the machine from the state x_i into the state x_j.
   Let S denote the subset of the states of a strongly connected machine. Let x_k denote the initial state. We denote by R_k the event that the subset S is in the initial state x_k.
   Show that $\bigcup_K R_K = ER$, where E is the universal event and R is some regular event.
7. Let $f(t)$ denote an integer-valued function such that $0 \leq f(t) \leq t$ and

$$\overline{\lim_{t \to \infty}} f(t) = \overline{\lim_{t \to \infty}} [t - f(t)] = \infty$$

   Show that the event "the number of symbols ρ_1 from the zeroth to the tth moment is equal to $f(t)$" is not regular.

8. Suppose that we are given a finite automaton A with initial state x^0. Let R denote a set of sequences at the input. Suppose that to each of these sequences is assigned a sequence in a set K of sequences of states. Show that, if R is regular, so is K. Does nonregularity of R imply nonregularity of K?

9. Suppose that we are given an s-machine with initial state x^0. At the input of this machine, sequences from the universal set E are applied. Show that the set of output sequences of the machine is regular.

10. Under the conditions of exercise 9, suppose that only sequences belonging to some regular set L are applied at the input. Is the assertion that the output sequences constitute a regular set valid in this case?

CHAPTER 8

1. Synthesize a finite automaton that represents by the appearance of the symbol λ_1 at the output the regular event

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

2. Do the same thing for

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

3. Synthesize a finite automaton representing the following definite event: the input sequence terminates with the symbols $\rho_1\rho_2$ or $\rho_3\rho_4\rho_1$. Write the regular expression for this event.

4. Synthesize the indicator of evenness of a discrete time moment. The regular expression corresponding to it has the form

$$R = [(\rho_1 \vee \rho_2 \vee \rho_3 \vee \rho_4)(\rho_1 \vee \rho_2 \vee \rho_3 \vee \rho_4)]^*.$$

CHAPTER 9

1. By using the algorithm of Aufenkamp and Hohn, show that the machine of Fig. 1 is minimal. The alphabets: $\rho = \{\rho_1, \rho_2\}$

$$x = \{x_1, x_2, x_3, x_4\} ; \lambda = \{\lambda_0, \lambda_1\}$$

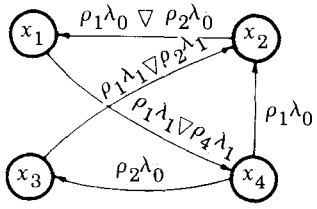


Fig. 1

2. Minimize the machine with interconnection matrix C by using the algorithm of Aufenkamp and Hohn:

$$C = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{array} \begin{array}{c} \begin{array}{cccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{array} \\ \begin{bmatrix} 0 & \rho_1\lambda_0 & 0 & \rho_2\lambda_1 & 0 & \rho_3\lambda_0 \\ \rho_2\lambda_0 & 0 & \rho_3\lambda_0 & 0 & \rho_1\lambda_1 & 0 \\ 0 & \rho_1\lambda_1 & 0 & \rho_3\lambda_0 & 0 & \rho_2\lambda_1 \\ \rho_2\lambda_0 & 0 & \rho_3\lambda_0 & 0 & \rho_1\lambda_1 & 0 \\ 0 & \rho_1\lambda_0 & \rho_2\lambda_0 & \rho_3\lambda_1 & 0 & 0 \\ \rho_3\lambda_1 & 0 & \rho_1\lambda_0 & 0 & \rho_2\lambda_0 & 0 \end{bmatrix} \end{array}$$

3. Show whether the following machine does or does not have equivalent states:

$$C = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{array} \begin{array}{c} \begin{array}{ccccc} x_1 & x_2 & x_3 & x_4 & x_5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & \rho_2\lambda_0 & \rho_1\lambda_1 \\ 0 & 0 & \rho_1\lambda_1 & 0 & \rho_2\lambda_0 \\ 0 & \rho_2\lambda_0 & \rho_1\lambda_1 & 0 & 0 \\ 0 & 0 & \rho_2\lambda_1 & \rho_1\lambda_1 & 0 \\ \rho_2\lambda_0 & 0 & 0 & \rho_1\lambda_1 & 0 \end{bmatrix} \end{array}$$

4. Minimize the following machine (with respect to strong equivalence):

$$C = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{array} \begin{array}{c} \begin{array}{cccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{array} \\ \begin{bmatrix} 0 & \rho_1\lambda_0 & 0 & \rho_2\lambda_0 & 0 & 0 \\ 0 & 0 & \rho_1\lambda_1 & 0 & 0 & \rho_2\lambda_0 \\ \rho_2\lambda_0 & 0 & 0 & \rho_1\lambda_0 & 0 & 0 \\ 0 & \rho_2\lambda_0 & 0 & 0 & 0 & \rho_1\lambda_0 \\ \rho_1\lambda_0 & 0 & \rho_2\lambda_0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \rho_1\lambda_1 \vee \rho_2\lambda_0 & 0 \end{bmatrix} \end{array}$$

5. From the interconnection matrix C draw a diagram of the states of the machine. Minimize it by using the algorithm of Aufenkamp and Hohn (strong equivalence).

$$C = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{array} \begin{array}{c} \begin{array}{cccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{array} \\ \begin{bmatrix} \rho_2\lambda_0 & \rho_1\lambda_1 & 0 & 0 & 0 & 0 \\ 0 & \rho_2\lambda_0 & \rho_1\lambda_0 & 0 & 0 & 0 \\ 0 & 0 & \rho_2\lambda_0 & \rho_1\lambda_0 & 0 & 0 \\ 0 & 0 & 0 & \rho_2\lambda_0 & \rho_1\lambda_1 & 0 \\ 0 & 0 & 0 & 0 & \rho_2\lambda_0 & \rho_1\lambda_0 \\ \rho_1\lambda_0 & 0 & 0 & 0 & 0 & \rho_2\lambda_0 \end{bmatrix} \end{array}$$

6. Show that the machine of Fig. 2 is minimal (strong equivalence). Find the groups of equivalent states in the case of the set of admissible input sequences L that contain all sequences of length 2.

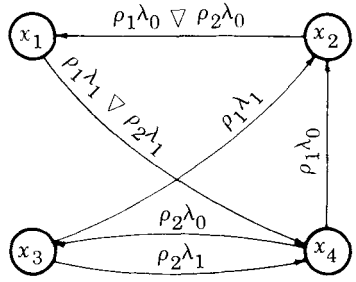


Fig. 2

7. Show whether the machine of Fig. 3 does or does not have equivalent states. For this machine, construct partitions of all states into equivalence classes with respect to input sequences of length 1, 2, 3, and 4.

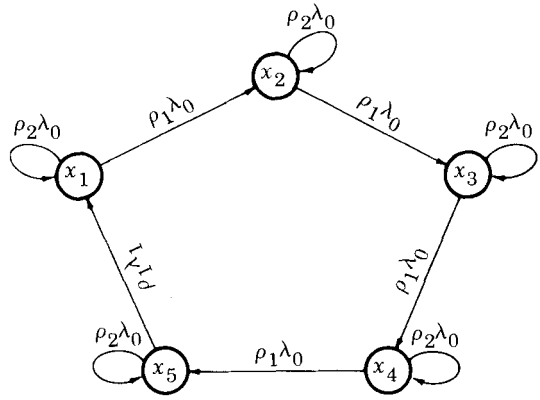


Fig. 3

8. Show whether the machine of Fig. 4 has equivalent states or not. Construct partitions into groups of equivalent states with respect to input sequences of length 1, 2, and 3.

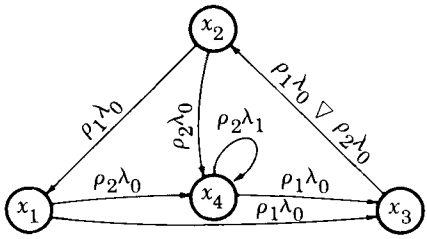


Fig. 4

9. The machine of Fig. 5 does not have equivalence states. However, if we take the definition of weak equivalence, corresponding to this machine is a minimal machine with three states. Construct it.

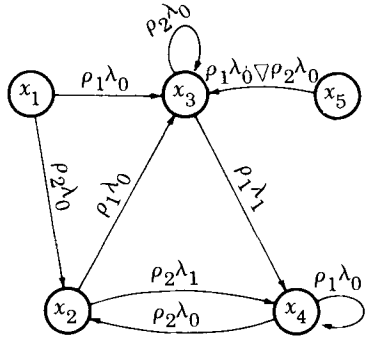


Fig. 5

10. Minimize the machine of Fig. 6 with respect to weak equivalence.

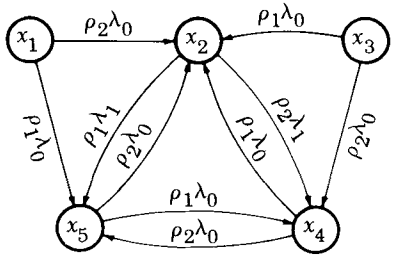
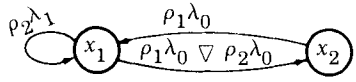


Fig. 6

11. Denote by E_l the set of all input sequences of length l. For arbitrary fixed l^*, construct for a given machine A (see Fig. 7) a machine B equivalent to A in the sense of weak equivalence with respect to E_{l^*} but not equivalent with respect to $E_{l^{**}}$ for $l^{**} > l^*$.



Machine A

Fig. 7

12. An s-machine is said to be strongly connected if, for every pair x_i and x_j of states of that machine there exists an input sequence that converts the machine from the state x_i to the state x_j. Show that, for strongly connected machines, weak equivalence of two machines implies their strong equivalence (so that in this case the concepts of strong and weak equivalence coincide.)

13. Show that, for completely determined automata (that is, automata without restrictions of the Aufenkamp type) for an arbitrary set of input sequences L and arbitrary x_i, x_j, x_k the equivalences with respect to L

$$x_i \sim x_j \ ; \ x_j \sim x_k$$

imply $x_i \sim x_k$ (transitivity of equivalence). Show that the number of groups of equivalent states with respect to L is the same in all machines that are pairwise equivalent with respect to L.

14. The set L contains a single sequence $\rho_1\rho_2$. Minimize the machine of Fig. 8 up to the 4th states with respect to L (strong equivalence)

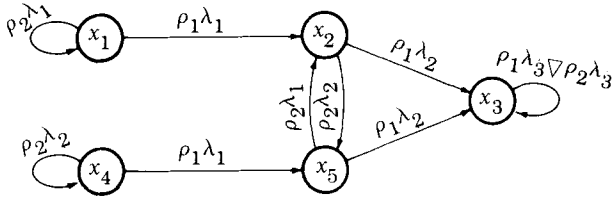


Fig. 8

(Hint: direct the arrow λ_1 from the state x_4x_5 to the state x_1 and remove the arrow $\rho_2\lambda_1$ from x_5 to x_2.)

15. The set L contains a single sequence $\rho_1\rho_2\rho_2$. Minimize up to the 4th states the machine of Fig. 9 with respect to L (strong equivalence).

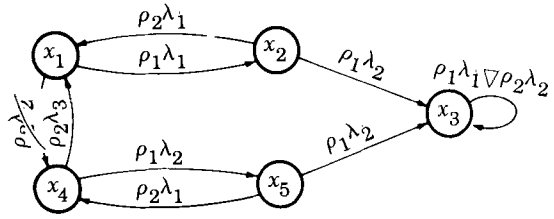


Fig. 9

16. Suppose that we are given the machine of Fig. 10 and the set L containing a single sequence $\rho_1\rho_2\rho_2$. In the machine the states x_2 and x_5 are equivalent with respect to L. However, it is imposible to minimize the machine of Fig. 10. Prove this. (Here, it is a question of minimization with respect to L in the sense of either strong or weak equivalence.)

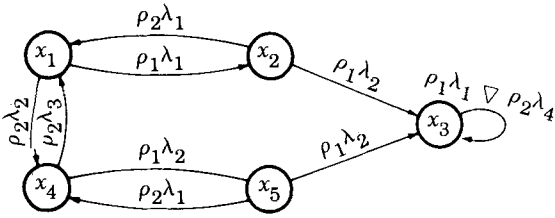


Fig. 10

17. The set L contains all sequences of input symbols such that a single symbol does not appear twice in a row. Minimize the machine of Fig. 11 with respect to L up to the 3rd states with respect to strong equivalence and up to the second states with respect to weak equivalence.

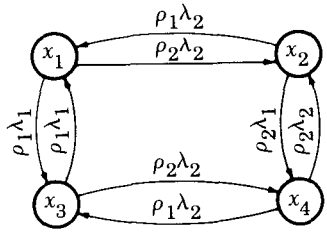


Fig. 11

18. Suppose that we are given the automaton of Fig. 12 with input alphabet $\{\rho_1, \rho_2, \rho_3\}$ and restrictions of the Aufenkamp type. Write the regular expressions of the set of admissible input sequences for the states x_1, x_2, x_3, x_4, and x_5

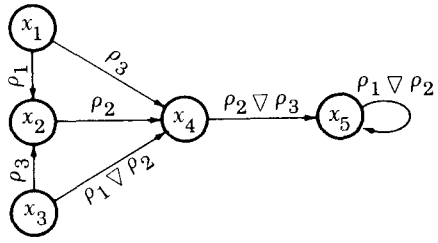


Fig. 12

19. Do the same thing as in exercise 18 for the automaton of Fig. 13.

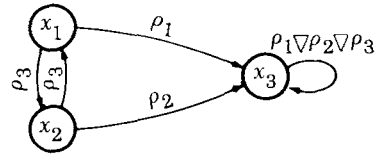


Fig. 13

Is the set of admissible input sequences for an arbitrary state x_i of the automaton with restrictions of the Aufenkamp type always regular?

20. Suppose that we are given an arbitrary s-machine with restrictions of the Aufenkamp camp that is in the initial state x_0. All sequences in the set L_{x_0} of sequences admissible for the state x_0 are applied at the input. Is the set of output sequences of this machine regular?

21. Is the theorem of regularity of representable events for an automaton with restrictions of the Aufenkamp type valid?

22. Minimize the s-machine of Fig. 14 with restrictions of the Aufenkamp type.

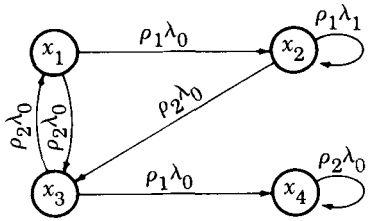


Fig. 14

23. To simplify the work, suppose that the following procedure is chosen for minimizing an s-machine with restrictions of the Aufenkamp type: First, we minimize the given machine in accordance with Aufenkamp's algorithm (symmetric partitioning into generalized i-matrices) and then we minimize the resulting machine by Hill's method. Does this approach guarantee construction of a minimal machine?

24. Figure 15 shows a diagram of the states of the s-machine. In that machine there are no restrictions of the Aufenkamp type (all transfers are determined) but the output transformer is undetermined: we do not know the value of λ for x_2 and ρ_1 (the loop in the diagram of the states). One can easily show that no extension of the definition of the output transformer will make it possible to minimize the machine of Fig. 15; that is, equivalence states do not arise.

However, it is possible to minimize this machine if we understand the word minimize in the following way: for a given s-machine M, it is required to construct an s-machine N such that the following two conditions are satisfied: (1) To every state x_i of the machine M there corresponds at least one state x_j of the machine N such that, for an arbitrary input sequence with the initial states x_i and x_j, the output sequence of the machine M coincides, wherever it is designed, with the output sequence of the machine N.

(2) No s-machine N' exists satisfying condition (1) with fewer states than the machine N.

Minimize the machine of Fig. 15 in accordance with this definition up to the 2nd states.

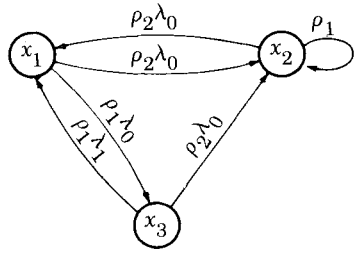


Fig. 15

CHAPTER 10

1. Suppose that we are given a slow machine G (see Fig. 1). Construct a minimal fast machine reproducing it under a clock-rate transformation with the clock rate determined by a change in the state at the input of the machine G.

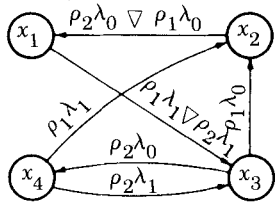


Fig. 1

2. Do the same thing as in problem 1 for the slow machine G of Fig. 2.

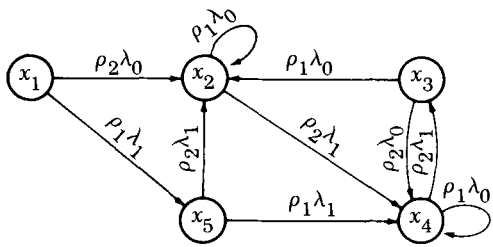


Fig. 2

In the following problems, the law of transformation of the clock rate is as follows: a slow moment occurs at the instant $\rho + 1$ if a regular event R occurs at the instant ρ, that is, the sequence $\rho_0\rho_1 \cdots \rho_\rho$ belongs to a regular set R.

3. Show that the machine S the diagram of the states of which is shown in Fig. 3 does not represent any slow machine if the law of transformation of clock rate is given by the regular set $R = (\rho_1^*\rho_2)^*$.

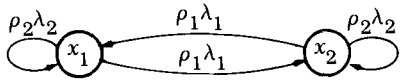


Fig. 3

4. A fast machine S is shown in Fig. 4. The law of transformation of clock rate is determined by a regular set R that can be represented in the automaton A (see Fig. 5) from the initial state a_0 by a set of states $\{a_0, a_1, a_2, a_3\}$. Construct a machine G that the machine S represents under such a transformation of clock rate.

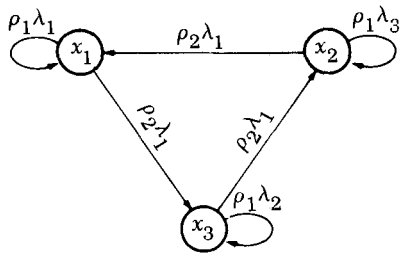


Fig. 4

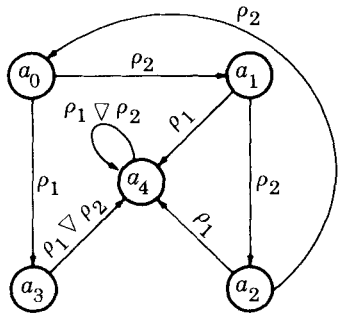


Fig. 5

5. Let S denote an arbitrary machine with initial state x_0 and let R denote a regular set defining a transformation of clock rate. Consider the set of all slow tapes obtained from fast ones as a result of the transformation of clock-rate. In this set, let us denote by λ_1 some event G_i (corresponding to the set of slow input sequences that lead to the occurence λ_i). Show that the set G_i is regular.

6. For the machine S shown in Fig. 4 and the machine G shown in Fig. 6, construct a regular set R such that the machine S reproduces the machine G under a transformation of clock rate determined by the set R.

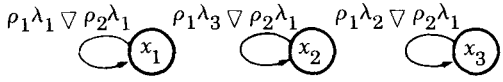


Fig. 6

(Hint: consider the following correspondence of events:

$$x_1^S \leftrightarrow x_1^G \; ; \; x_2^S \leftrightarrow x_2^G \; ; \; x_3^S \leftrightarrow x_3^G \; .)$$

7. Let S and G denote machines. Let x^G denote a state of the machine G. What conditions must the machine S satisfy for it to be possible to construct a regular set R such that corresponding to the state x^G there will be a state x^S of the machine S in the sense of reproduction.

CHAPTER 11

1. Suppose that Fig. 1 is the diagram of states of a strongly connected s-machine S.

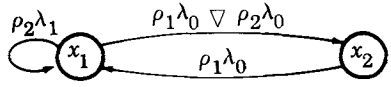


Fig. 1

Construct a diagram of states of the s-machine N for which the result of an experiment of length q coincides with the result of the experiment with the given machine S for arbitrary initial states of the s-machine N. (this last condition differs from the condition of the example given on page 399 of original Russian, Figs. 11.1 and 11.2).

2. Suppose that Fig. 2 is the diagram of states of an s-machine. Find the shortest experiment determining the last state of this machine under the condition that all the states can be initial.

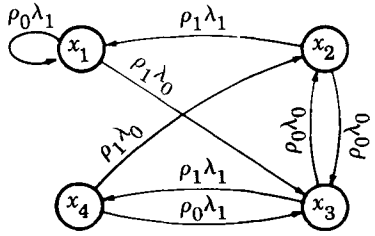


Fig. 2

3. Show that the estimates (11.17) and (11.18) are exact for arbitrary N. To do this, construct diagrams of the states of an s-machine and a finite automaton with N states. To determine the last states of these, it is necessary to make experiments the lengths of which are determined by these estimates.

4. Figure 3 shows the diagrams of the states of three finite automata. Show that it is possible to single out any one of them by an experiment of length not exceeding 4.

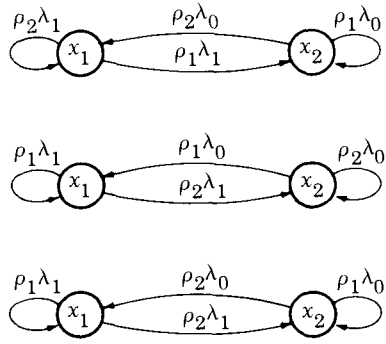


Fig. 3

5. Find an experiment with the aid of which we can ascertain the structure of any one of four finite automata shown in Fig. 4, that is, we can run through all states and all arrows of any one of these automata if the state κ_2 is the initial state.

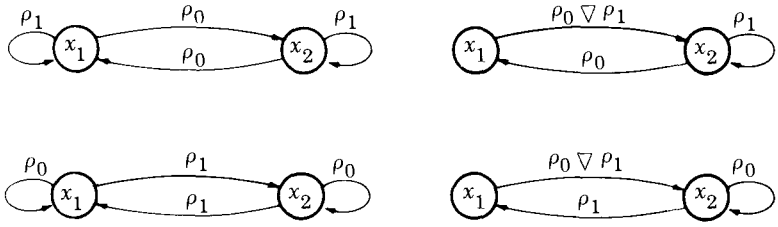


Fig. 4

6. Construct the diagram of the states of a strongly connected s-machine with N states to which it necessary to apply an input sequence of length $N^2/4$ in order to go through all its states. Is the quantity $N^2/4$ maximum for the length of the experiment as a result of which an arbitrary strongly connected machine goes through all states or is it possible to find a strongly connected machine for which an experiment of greater length is required if the machine is to go through all states?

7. Suppose that we are given the set of all strongly connected machines with N states. Show that there exists an experiment making it possible to go through all states of any one of these

machines independently of its initial state and estimate the length of the experiment.

8. Figure 5 shows the diagram of the states of a strongly connected s-machine with four nonequivalent states. One can easily show that it is possible to set up the following one-to-one correspondence between the results of experiments and states into which the s-machine goes at the end of these experiments.

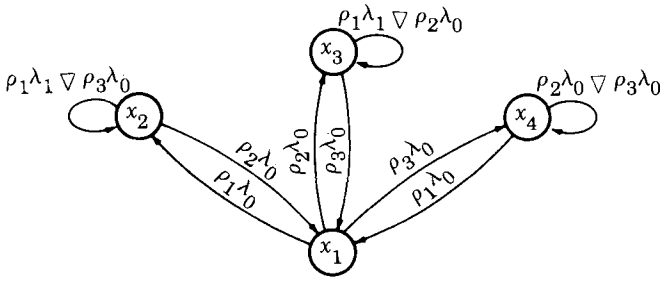
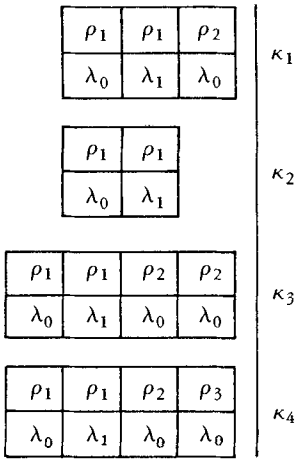


Fig. 5

The result corresponds to a transfer of the s-machine into the state



Try to show that for a given strongly connected s-machine with N nonequivalent states, then to each of its states x_i it

is possible to assign an experiment of length not exceeding N the result of which indicates unambiguously that the machine has gone into just that state x_i. (In this example, this estimate is attained.)

9. Suppose that we are given a set of 2^n strongly connected finite automata of the form shown in Fig. 6 (locks).

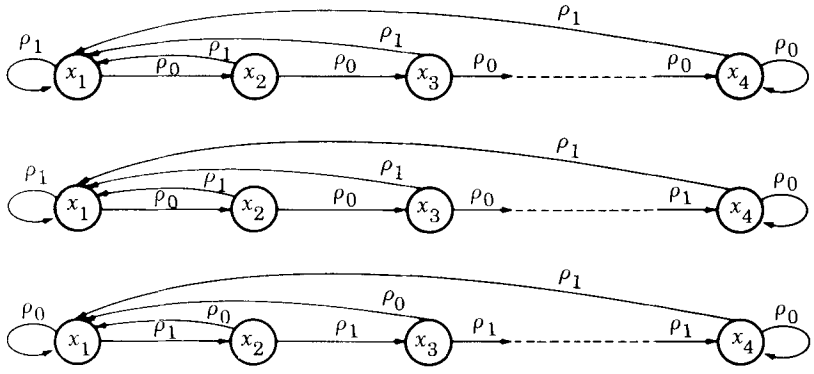


Fig. 6

At the beginning of the experiment, the finite automata are in the state κ_1. Show that an upper bound q of the length of the experiment necessary for attaining the state κ_n of any one of these automata is determined by the equation $q = 2^n \ln n$.

Bibliography

1. AIZERMAN, M. A., GUSEV, L. A., ROZONOER, L. I., SMIRNOVA, I. M. and TAL', A. A., Finite Automata, 1. Avtomatika i Telemekhanika, Vol. 21, No. 2, 1960.
2. AIZERMAN, M. A. et al., Finite Automata, 2. Avtomatika i Telemekhanika, Vol. 21, No. 3, 1960.
3. AIZERMAN, M. A. et al., Methods of realization of a finite automaton whose rhythm is determined by variation of the input state, Avtomatika i Telemekhanika, Vol. 21, No. 12, 1960.
4. AIZERMAN, M. A. et al., Algorithmic unsolvability of the recognition problem relating to representation of recursive events in finite automata, Avtomatika i Telemekhanika, Vol. 22, No. 6, 1961.
5. AUFENKAMP, D. D., Analysis of sequential machines, II. Matematika (a periodic collection of translations of foreign articles) 3:6, 1959.
6. AUFENKAMP, D. D. and HOHN, F. E., Analysis of sequential machines, I. Matematika (a periodic collection of translations of foreign articles) 3:3, 1959.
7. AUFENKAMP, D. D., HOHN, F. E. and SECHU, S., Theory of nets, Matematika (a periodic collection of translations of foreign articles) 3:3, 1959.
8. BAZILEVSKY, Yu. Ya., Problems of the theory of temporal logical functions, Voprosy Teorii Matemat. Machin (a collection), No. 1, Fizmatgiz Press, 1958.
9. BAZILEVSKY, Yu. Ya., Some transformations of finite automata. Theory and Application of Discrete Automatic Systems (a collection), Ya. Z. Tsypkina (ed.), Acad. Sci. Press, Moscow, 1960.
10. BAZILEVSKY, Yu. Ya., Solution of temporal logical equations by the reduction method. A Collection of Works of the Conference on the Theory and Application of Discrete Automatic Systems, Acad. Sci. Press, Moscow, 1960.
11. BERKELEY, E., Symbolic Logic and Intelligent Machines, Reinhold Publishers, N. Y., 1959; Berkeley Enterprises, Inc., Newtonville, Mass., 1959.

12. BERENDS, T. K. and TAL', A. A., Pneumatic relay schemes, Avtomatika i Telemekhanika, Vol. 20, No. 11, 1959.
13. BLOCH, A. Sh., Synthesis of multistage circuits, Sbornik Trudov Instituta Mashinovedeniya i Avtomatizatsii AN BSSR, No. 1, 1961.
14. BLOCH, A. Sh., A canonical method for the synthesis of electronic circuits. Sbornik Trudov Instituta Mashinovedeniya i Avtomatizatsii AN BSSR, No. 1, 1961.
15. BLOCH, A. Sh., The synthesis of relay contact circuits, Doklady Akad. Nauk SSSR, Vol. 117, No. 4, 1957.
16. BLOCH, A. Sh., Problems solvable by sequential machines, Problemy Kibernetiki (a collection), No. 3, Fizmatgiz Press, Moscow, 1960.
17. BLOCH, A. Sh., A canonical method for the synthesis of contact circuits, Avtomatika i Telemekhanika, Vol. 22, No. 6, 1961.
18. WANG HAO and McNAUGHTON, R., Axiomatic Systems in Set Theory, IL, Moscow, 1963.
19. VAVILOV, E. N., and OSINSKY, L. M., A method for the structural synthesis of finite automata, Avtomatika, No. 2. 1963.
20. GAVRILOV, M. A., The present state of the art and fundamental trends in the development of the theory of relay circuits, Papers on the All-Union Conference on the Theory of Relay Devices, (a collection), Acad. Sci. Press, Moscow, 1957.
21. GAVRILOV, M. A., The Theory of Relay Contact Circuits, Acad. Sci. Press, Moscow-Leningrad, 1950.
22. GAVRILOV, M. A., The structural theory of relay devices (a lecture). VZEI, part 1, 1959; part 2, 1960; part 3, 1961. VZEI Press.
23. GAVRILOV, M. A., Minimization of Boolean functions characterizing relay networks, Avtomatika i Telemekhanika, Vol. 20, No. 9, 1959.
24. HILBERT, D., and ACKERMANN, W., Principles of Mathematical Logic, Chelsea Publishing Co., N. Y.
25. GINSBURG, S., The length of the shortest experiment determining the terminal states of a machine, Kiberneticheskiy Sbornik No. 3, IL, Moscow, 1961.
26. GLEBSKY, Yu. V., Coding with the aid of automata with a finite internal memory, Problemy Kibernetiki (a collection), No. 7, Fizmatgiz Press, Moscow, 1962.
27. GLUSHKOV, V. M., Abstract theory of automata, Uspekhi Matem. Nauk, Vol. 16, No. 5 (10), 1961; Vol. 17, No. 2 (104), 1962.

28. GLUSHKOV, V. M., Some synthesis problems of digital automata, Zh. Vychislit. Matemat. i Matem. Fiz., Vol. 1, No. 3, May-June, 1961.
29. GLUSHKOV, V. M., Abstract automata and splitting of the free semigroups, Doklady Akad. Nauk SSSR, Vol. 136, No. 4, 1961.
30. GOODSTEIN, R. L., Mathematical Logic, A collection, edited and prefaced by S. A. Yanovskaya, IL, Moscow, 1961.
31. DAVIS, M. D., A note on universal Turing machines, Automata (a collection), edited by C. E. Shannon and J. McCarthy, IL, Moscow, 1956; see also: Shannon, C. E. and J. McCarthy, Automata Studies, Princeton University Press, Princeton, N. J., 1956.
32. ZHURAVLEV, Yu. I., Various concepts of the minimality of disjunctive normal forms, Sibirsky Matem. Zh., Vol. 1, No. 4, November-December, 1960.
33. ZHURAVLEV, Yu. I., Separability of subsets of the vertices of an n-dimensional unit cube, Trudy Matem. In-ta im. V. A. Steklova, Vol. 51, Acad. Sci. Press, 1958.
34. ZAVOLOKINA, Z. I., Magnetic Elements in Digital Computers, Gosenergoizdat Press, Moscow-Leningrad, 1958.
35. ZAKREVSKIY, A. D., Minimization of structural formulas of multistage circuits, Trudy Sibirskogo Fiziko-Tekhnicheskogo In-ta pri Tomskom Gos. Universitete im. Kuybysheva, No. 40, 1961.
36. ZAKREVSKIY, A. D., The synthesis of sequential automata, Trudy Sibirskogo Fiziko-Tekhnicheskogo Instituta No. 40, 1961.
37. KAZAKOV, V. D., and KUZNETSOV, O. P., List of foreign works on the theory of relay action devices and finite automata for the year 1958, Avtomatika i Telemekhanika, Vol. 21, No. 9, 1960; List of Soviet works on the theory of relay circuits and finite automata for the year 1959, Avtomatika i Telemekhanika, Vol. 22, No. 2, 1961; Vol. 24, No. 5, 1963.
38. KOL'MAN, E., The significance of symbolic logic, Studies in Logic. A collection edited by E. Kol'man et al., Acad. Sci. Press, Moscow, 1959.
39. CULBERTSON, J. T., Some uneconomical robots, Automata. A collection edited by C. E. Shannon and J. McCarthy, IL, Moscow, 1956; see also Culbertson, J. T., Minds of Robots, U. of Illinois Press, Urbana, Ill., 1963; also: Shannon, C. E. and J. McCarthy, Automata Studies, Princeton University Press, Princeton, N. J., 1956.
40. KARATSUBA, A. A., Solution of a problem of the theory of finite automata, Uspekhi Matem. Nauk., Vol. 15, No. 3, 1960.

41. KLEENE, S. C., Representation of events in nerve nets and finite automata, Automata. A collection edited by C. E. Shannon and J. McCarthy, IL, Moscow, 1956; see also: Shannon, C. E. and J. McCarthy, Automata Studies, Princeton University Press, Princeton, N. J., 1956.
42. KLEENE, S. C., Introduction to Metamathematics, D. Van Nostrand Publishers, N. Y., 1952.
43. KOBRINSKY, N. E., and TRAKHTENBROT, B. A., The construction of a general theory of logical networks. Studies in Logic. A collection edited by E. Kol'man et al., Acad. Sci. Press, Moscow, 1959.
44. KOBRINSKY, N. E., and TRAKHTENBROT, B. A., Introduction to the theory of Finite Automata, Fizmatgiz Press, Moscow, 1961.
45. CALDWELL, S., Logical Synthesis of Relay Devices, IL, Moscow, 1961; see Caldwell, S. H., Switching Circuits and Logical Design, John Wiley & Sons, N. Y., 1958.
46. COPE, I. M., ELGOT, C. C., and WRIGHT, J. B., Realization of events in logical nets, Kiberneticheskiy Sbornik, No. 3, IL, Moscow, 1961.
47. KUDRYAVTSEV, V. B., Completeness theorem for one class of loop-free automata, Doklady Akad. Nauk SSSR, Vol. 132, No. 2, 1960.
48. KUDRYAVTSEV, V. B., Problems of completeness relating to systems of automata, Doklady Akad. Nauk SSSR, Vol. 130, No. 6, 1960.
49. KAZAREV, V. G., and PIYL', E. I., A method for the synthesis of finite automata, Avtomatika i Telemekhanika, Vol. 22, No. 9, 1961.
50. LAZAREV, V. G., A method for determination of the number of relays necessary for the construction of relay circuits from given operation conditions, Problemy Peredachi Informatsii, No. 1, Acad. Sci. Press, 1959.
51. LAZAREV, V. G., Determination of the minimal number of intermediate relays in the synthesis of multistage circuits, Sbornik rabot po provodnoy svyazi, No. 5, Acad. Sci. Press, 1956.
52. LAZAREV, V. G. and PIYL', E. I., Reduction of the number of states in one class of finite automata, Doklady Akad. Nauk SSSR, Vol. 143, No. 5, 1962.
53. LETICHEVSKY, A. A., Synthesis of finite automata, Doklady Akad. Nauk Ukr SSR, Vol. 11, No. 2, 1961.
54. LETICHEVSKY, A. A., Conditions of completeness for finite automata, Zh. Vychislit. Matemat. i Matem. Fiz., Vol. 1, No. 4, 1961.

55. LUNTS, A. G., Application of matrix Boolean algebra to the analysis and synthesis of relay contact circuits, Doklady Akad. Nauk SSSR, Vol. 70, No. 3, 1950.
56. LUPANOV, O. B., Asymptotic evaluation of the number of graphs and networks with n edges, Problemy Kibernetiki (a collection), No. 4, Fizmatgiz Press, Moscow, 1960.
57. LUPANOV, O. B., A method for circuit synthesis, Izvestiya Vuzov, Radiofizika, No. 1, 1958.
58. LUPANOV, O. B., The possibilities of synthesis of circuits from arbitrary elements, Trudy Matem. Instituta im. V. A. Steklova, Vol. 51, Acad. Sci. Press, 1958.
59. LUPANOV, O. B., The possibilities of synthesis of circuits from diverse elements, Doklady Akad. Nauk SSSR, Vol. 103, No. 4, 1955.
60. LUPANOV, O. B., Realization of the algebra of logic functions by finite class formulas (by formulas of bounded depth on the basis of &, V, —), Problemy Kibernetiki (a collection), No. 6, 1961.
61. LUPANOV, O. B., The principle of local coding and realization of functions of certain classes by circuits of functional elements, Doklady Akad. Nauk SSSR, Vol. 140, No. 2, 1961.
62. McCULLOCH, W. S., and PITTS, W., A logical calculus of the ideas related to nervous activity, Automata. A collection edited by C. E. Shannon and J. McCarthy, IL, Moscow, 1956; see also: Shannon, C. E. and J. McCarthy, Automata Studies, Princeton University Press, Princeton, N. J., 1956.
63. McCARTHY, J., The inversion of functions defined by Turing machines, Automata. A collection edited by C. E. Shannon and J. McCarthy, IL, Moscow, 1956; see also: Shannon, C. E. and J. McCarthy, Automata Studies, Princeton University Press, Princeton, N. J., 1956.
64. MARKOV, A. A., The algorithm theory, Trudy Matem. Instituta im. V. A. Steklova, Vol. 42, 1954.
65. MARKOV, A. A., Mathematical logic and computer mathematics, Vestnik AN SSSR, No. 8, 1957.
66. MARTYNYUK, V. V., Relationship between the memory and certain potentialities of a finite automaton, Problemy Kibernetiki (a collection), No. 5, Fizmatgiz Press, Moscow, 1961.
67. MEDVEDEV, Yu. T., A class of events representable in a finite automaton, Supplement to Automata. A collection edited by C. E. Shannon and J. McCarthy, IL, Moscow, 1956; see Shannon, C. E. and J. McCarthy, Automata Studies, Princeton University Press, Princeton, N. J., 1956.

68. MEKLER, Ya. I., Simplification of the algebraic synthesis of relay circuits, Avtomatika i Telemekhanika, Vol. 19, No. 12, 1958.
69. MINSKIY, M. L., Some universal elements for finite automata, Supplement to Automata. A collection edited by C. E. Shannon and J. McCarthy, IL, Moscow, 1956; see Shannon, C. E. and J. McCarthy, Automata Studies, Princeton University Press, Princeton, N. J., 1956.
70. MOISIL, G. C., Algebraic theory of the operation of real relay contact circuits, Papers on the All-Union Conference on the Theory of Relay Devices (a collection), Acad. Sci. Press, 1957.
71. MOISIL, G. C., and IOANIN, G., Synthesis of relay contact circuits for given operation conditions of the executive elements, Zh. Chistoy i Priklad. Matemat., Akad. RNR, Vol. 1, No. 2, 1956.
72. MOORE, E. F., Gedanken-experiments on sequential machines, Automata. A collection edited by C. E. Shannon and J. McCarthy, IL, Moscow, 1956; see Shannon, C. E. and J. McCarthy, Automata Studies, Princeton University Press, Princeton, N. J., 1956.
73. NEUMANN (von) J., Probabilistic logics and the synthesis of reliable organisms from unreliable components, Automata. A collection edited by C. E. Shannon and J. McCarthy, IL, Moscow, 1956; see Shannon, C. E. and J. McCarthy, Automata Studies, Princeton University Press, Princeton, N. J., 1956.
74. NECHIPORUK, E. I., The synthesis of *R*-circuits, Doklady Akad. Nauk SSSR, Vol. 137, No. 5, 1961.
75. NOVIKOV, P. S., Elements of Mathematical Logic, Fizmatgiz Press, Moscow, 1959.
76. OSTIANU, V. M., and TOMFEL'D, Yu. L., One application of mathematical logic, Uch. Zapiski Kishinevskogo Gos. Un-ta, Vol. 29, 1957.
77. PETER, R., Recursive Functions, Pergamon Press, N. Y.
78. POVAROV, G. N., The logical synthesis of electronic computer and control circuits. Studies in Logic, edited by E. Kol'man et al., Acad. Sci. Press, 1959.
79. POVAROV, G. N., The symmetry of Boolean functions, Trudy Vsesoyuznogo Matematicheskogo s''yezda, Vol. 4 (a brief summary of the papers), Acad. Sci. Press, 1959.
80. POVAROV, G. N., Functional separability of Boolean functions, Doklady Akad. Nauk SSSR, Vol. 94, No. 5, 1954.
81. POVAROV, G. N., The study of symmetrical Boolean functions from the viewpoint of relay contact circuits, Doklady Akad. Nauk SSSR, Vol. 104, No. 2, 1955.

82. POVAROV, G. N., The structural theory of communication networks, Problemy Peredachi Informatsii (a collection), No. 1, Acad. Sci. Press, 1959.
83. PARKHOMENKO, P. P., Block analyzer of relay circuits, Tekhnicheskiye Sredstva Avtomatizatsii i Telemekhaniki (a collection), No. 14, Moscow, 1961.
84. POPOVICH, K., Minimal disjunctive form of Boolean functions, Papers on the All-Union Conference on the Theory of Relay Devices (a collection), Acad. Sci. Press, 1957.
85. POSPELOV, D. A., Arithmetical and Logical Principles of Digital Computers, Part I: Arithmetical and logical principles of digital computers; Part II: Algebra of logic functions, the synthesis and analysis of circuits having time-independent operation, MEI Press, Moscow, 1960.
86. Use of Logic in Science and Technology. A collection edited by P. V. Tavanets et al., Acad. Sci. Press, 1960.
87. ROGINSKIY, V. N., Elements of the Structural Synthesis of Relay Control Circuits, Acad. Sci. Press, 1959.
88. ROGINSKIY, V. N., Equivalent transformations of relay circuits, Sbornik nauchnykh rabot po provodnoy svyazi, No. 6, 1957.
89. ROGINSKIY, V. N., A graphical method for the construction of contact circuits, Elektrosvyaz', No. 11, 1957.
90. RANI, J. N., Sequential functions, Kiberneticheskiy Sbornik, No. 3, IL, Moscow, 1961.
91. A collection of articles on mathematical logic and its application to certain problems of cybernetics, Trudy Matem. Instituta im. V. A. Steklova, Vol. 51, Acad. Sci. Press, 1958.
92. SKORNYAKOV, L. A., A class of automata (neutral nets) Problemy Kibernetiki (a collection), No. 4, Fizmatgiz Press, Moscow, 1960.
93. SOKOLOV, O. B., The construction of functionally stable automata, Summary of 1960 scientific conference of the Kazan' State University, Kazan', 1960.
94. SORKIN, Yu. I., Algorithmic solvability of the isomorphism problem in relation to automata, Doklady Akad. Nauk SSSR, Vol. 137, No. 4, 1961.
95. STOGNIY, A. A., The synthesis of an abstract automaton from the event representable by it, Zh. Vychislit. Matemat. i Matem. Fiz., Vol. 1, No. 3, 1961.
96. TARSKI, A., Introduction to Logic and to the Methodology of Deductive Sciences, 2nd ed. rev. 1946; 3rd ed. Oxford University Press.
97. TRAKHTENBROT, B. A., Algorithms and Mechanical Solution of Problems, Fizmatgiz Press, Moscow, 1957.

98. TRAKHTENBROT, B. A., Operators realizable in logic nets, Doklady Akad. Nauk SSSR, Vol. 112, No. 6, 1957.
99. TRAKHTENBROT, B. A., The synthesis of logical nets whose operators are described by means of one-place predicate calculus, Doklady Akad. Nauk SSSR, Vol. 118, No. 4, 1958.
100. TRAKHTENBROT, B. A., Asymptotic estimate of the complexity of logical nets with a memory, Doklady Akad. Nauk SSSR, Vol. 127, No. 2, 1959.
101. TRAKHTENBROT, B. A., Some constructions in one-place predicate logic, Doklady Akad. Nauk SSSR, Vol. 138, No. 2, 1961.
102. TRAKHTENBROT, B. A., Finite automata and one-place predicate logic, Doklady Akad. Nauk SSSR, Vol. 140, No. 2, 1961.
103. USPENSKIY, B. A., Lectures on Computable Functions, Fizmatgiz Press, Moscow, 1960.
104. FEL'DBAUM, A. A., Computers in Automatic Systems, Fizmatgiz Press, Moscow, 1959.
105. KHARKEVICH, A. D., Commutational circuits and their logic. Studies in Logic (a collection), Acad. Sci. Press, 1959.
106. TSEYTIN, G. S., Algorithmic operators in constructive metric spaces, Trudy Matem. In-ta im. V. A. Steklova, Vol. 67, 1962.
107. TSETLIN, M. L., Imprimitive circuits, Problemy Kibernetiki (a collection), No. 1, Fizmatgiz, 1958.
108. TSETLIN, M. L., Some problems relating to the behavior of finite automata, Doklady Akad. Nauk SSSR, Vol. 139, No. 4, 1961.
109. CHURCH, A., Introduction to Mathematical Logic, Princeton University Press, Princeton, N. J., 1956.
110. SHANNON, C. E., A universal Turing machine with two internal states, Automata. A collection edited by C. E. Shannon and J. McCarthy, IL, Moscow, 1956; see Shannon, C. E. and J. McCarthy, Automata Studies, Princeton University Press, Princeton, N. J., 1956.
111. SHESTAKOV, V. I., An algebraic method for the analysis of autonomous systems of two-position relays, Avtomatika i Telemekhanika, Vol. 15, No. 2, 1954; Vol. 15, No. 4, 1954.
112. SHESTAKOV, V. I., Vectorial-algebraic method for the analysis and synthesis of multistage relay circuits, Trudy III Vsesoyuznogo Matematicheskogo s''yezda, Vol. 1, 1956.
113. SHREYDER, Yu. A., The problem of dynamic planning and automata, Problemy Kibernetiki (a collection), No. 5, Fizmatgiz Press, Moscow, 1961.

114. ASHBY, W. R., Introduction to Cybernetics, John Wiley and Sons, N. Y., 1956.
115. YABLONSKIY, S. V., Functional constructions in many-valued logics, Trudy III Vsesoyuznogo Matematicheskogo S''yezda, Vol. 2, Acad. Sci. Press, 1956.
116. YABLONSKIY, S. V., On boundary logics, Doklady Akad. Nauk SSSR, Vol. 118, No. 4, 1958.
117. YABLONSKIY, S. V., Algorithmic difficulties in the synthesis of minimal contact circuits, Problemy Kibernetiki (a collection), No. 2, Fizmatgiz Press, Moscow, 1959.
118. YANOVSKAYA, S. A., Some features of mathematical logic and its relation to technical applications. Use of Logic in Science and Technology (a collection), Acad. Sci. Press, 1960.
119. YANOVSKAYA, S. A., Mathematical logic and the foundations of mathematics. Mathematics in USSR over 40 Years (1917-1957) [a collection], Vol. 1, Fizmatgiz Press, Moscow, 1959.
120. ABHYANKAR, S., Minimal "sum of products of sum" expression of Boolean functions, IRE Trans., Vol. EC-7, No. 4, 1958.
121. ABHYANKAR, S., Absolute minimal expressions of Boolean functions, IRE Trans., Vol. EC-8, No. 1, 1959.
122. ARANT, G. W., A time-sequential tabular analysis of flip-flop logical operation, IRE Trans., Vol. EC-6, June, 1957.
123. ARBIB, M., Turing machines, finite automata and neural nets, J. Assoc. Comp. Machines, Vol. 8, No. 4, 1961.
124. ARDEN, D., Delayed logic and finite state machines. Symposium on Switching Circuit Theory and Logical Design, AIEE, Detroit, Michigan, October, 1961.
125. AUFENKAMP, D. D., and HOHN, F. E., Oriented graphs and sequential machines, Bell Laboratories Reports, 30, 1954.
126. BALDA, M., Classification of automata, Automatizace, Vol. 4, No. 2, 1961.
127. BEATSON, T. J., Minimization of components in electronic switching circuits, Communication and Electronics, No. 4, July, 1958.
128. BELLMAN, R., Sequential machines ambiguity and dynamic programming, J. Assoc. Comp. Machines, Vol. 7, No. 1, 1960.
129. BELLMAN, R., Adaptive control processes. A guided tour. Princeton, New Jersey, 1961.
130. BELLMAN, R., HOLLAND, J., and KALABA, R., On an application of dynamic programming to the synthesis of logical systems, J. Assoc. Comp. Machines, Vol. 6, No. 4, 1959.
131. BIANCHII, R., and FREIMAN, C., On internal variable assignment for sequential switching circuits, IRE Trans., Vol. EC-10, No. 1, March, 1961.

132. BÜCHI, J. R., Weak second-order arithmetic and finite automata, Zeitschrift für Mathematische Logik und Grundlagen der Math., Vol. 6, No. 1, 1960.
133. BÜCHI, J. R., ELGOT, C. C., and WRIGHT, J. B., The nonexistence of certain algorithms of finite automata theory (Abstracts). Notices Amer. Math. Soc., Vol. 5, 1958.
134. BURKS, A. W., and WANG, H., The logic of automata, parts I-II, J. Assoc. Comp. Machines, Vol. 4, 1957.
135. BURKS, A. W., The logic of fixed and growing automata, Intern. Symp. on the Theory of Switching, Vol. 29, part 1, 1959.
136. CADDEN, W. J., Equivalent sequential circuits, IRE Trans., Vol. CT-6, No. 1, 1959.
137. CHURCH, A., Application of recursive arithmetic in the theory of computers and automata, Lecture Notes, Summer Conference, University of Michigan, June, 1958.
138. CHU, J. T., Some methods for simplifying switching circuits using don't care conditions, J. Assoc. Comp. Machines, Vol. 8, No. 4, 1961.
139. CULBERTSON, J. T., Robots and automata; a short history. Parts I-II. Computers and Automation, Vol. 6, No. 3-4, 1957.
140. CULBERTSON, J. T., Mathematics and logic for digital devices, Princeton Univ. Press, Princeton, N. J., 1958.
141. ČULIK, K., Some notes on finite-state languages and events represented by finite automata using labelled graphs, Casopis pro pěst. Mot., Vol. 86, No. 1, 1961.
142. DAVIS, M. D., Computability and Unsolvability, N. Y., 1957.
143. ELGOT, C. C., Decision problems of finite automata design and related arithmetics, Trans. Amer. Math. Soc., Vol. 98, No. 4, 1961.
144. ELGOT, C. C., Lectures on switching and automata theory. The University of Michigan Research Institute, Technical Report, January, 1959.
145. ELGOT, C. C., and RUTLEDGE, J. D., Operations on finite automata. Symposium on Switching Circuit Theory and Logical Design, AIEE, Detroit, Michigan, October, 1961.
146. ELSPAS, B., The theory of autonomous linear sequential networks, IRE Trans., Vol. CT-6, No. 1, 1959.
147. FITCH, F. B., Representation of sequential circuit in combinatory logic, Philosophy, Vol. 25, No. 4, October, 1958.
148. GEORGE, F. H., Behaviour network systems for finite automata. Methodos, Vol. 9, No. 35-37, 1957.
149. GILL, A., Introduction to the Theory of Finite-State Machines, McGraw-Hill Book Co., N. Y., 1962.

150. GILL, A., Comparison of finite-state models, IRE Trans., Vol. CT-7, No. 2, June, 1960.
151. GILL, A., State-indentification experiments in finite automata, Information and Control, Vol. 4, No. 23, Sept., 1961.
152. GILL, A., Cascaded finite state machines, IRE Trans., Vol. EC-10, No. 3, Sept., 1961.
153. GINSBURG, S., Compatibility of states of input independent machines, J. Assoc. Comp. Machines, Vol. 8, No. 3, July, 1961.
154. GINSBURG, S., On the reduction of superfluous states in a sequential machine, J. Assoc. Comp. Machines, Vol. 6, April, 1959.
155. GINSBURG, S., Connective properties preserved in minimal state machines, J. Assoc. Comp. Machines, Vol. 7, No. 4, 1960.
156. GINSBURG, S., A technique for the reduction of a given machine to a minimal state machine, IRE Trans., EC-8, No. 3, 1959.
157. GINSBURG, S., Sets of tapes accepted by different types of automata, J. Assoc. Comp. Machines, Vol. 8, No. 1, January, 1961.
158. GINSBURG, S., Some remarks on abstract machines, Trans. Amer. Math. Soc., Vol. 96, No. 3, 1960.
159. HAKIMI, S. L., On realizability of a set of trees, IRE Trans., Vol. CT-8, No. 1, March, 1961.
160. HARING, D. R., The sequential transmission expression for flow-graphs, MIT Electronics Systems Laboratory Technical Memorandum, No. 7848-T-14-3, November, 1960.
161. HARRIS, B., An algorithm for determining minimal representation of a logic function, IRE Trans., EC-6, No. 2, 1957.
162. HIBBARD, T. N., Least upper bound on minimal terminal state experiments for two classes of sequential machines, J. Assoc. Comp. Machines, Vol. 8, No. 4, October, 1961.
163. HIGONNET, R. A., and GREA, R. A., Logical Design of Electrical Circuits, McGraw-Hill Book Co., N. Y., 1958.
164. HILTON, A. M., Logic and switching circuits, Electric Manufacturing, April, 1960.
165. HOFFMAN, A. J., and SINGLETON, R. P., On Moore graphs with diameter 2 & 3, IBM Journ. of Research and Develop., Vol. 4, No. 5, 1960.
166. HOHN, F. E., Some mathematical aspects of switching, Amer. Mathem. Monthly, Vol. 62, 1955.
167. HOHN, F. E., Applied Boolean Algebra. An Elementary Introduction, McMillan Publ., N. Y., 1960.
168. HOLBROOK, E. L. Pneumatic logic. I-II. Control Engineering, Vol. 8, No. 7-8, 1961.

169. HOLLAND, L., Cycles and automata behaviour. Preliminary report. Session on Advanced Theory of Logical Design of Digital Computers, Univ. of Michigan, June, 1958.
170. HUFFMAN, D. A., The synthesis of sequential switching circuits, Journ. Franklin Inst., Vol. 257, No. 3-4, 1954.
171. HUFFMAN, D. A., The design and use of hazard-free switching networks, J. Assoc. Comp. Machines, Vol. 4, January, 1957.
172. HUFFMAN, D. A., Sequential transducer. Papers Buyer's Guide, IRE Trans., Vol. CT-6, No. 1, March, 1959.
173. HUMPHREY, W. S., Switching Circuits with Computer Applications, John Wiley and Sons, N. Y., 1958.
174. HUZINO, S., On some sequential machines and experiments, Mem. Fac. Sci. Kyusyu University, Vol. A12, No. 2, 1958.
175. HUZINO, S., Reduction theorems on sequential machines, Mem. Fac. Sci. Kyusyu Univ., Vol. A12, No. 2, 1958.
176. HUZINO, S., On the existence of Scheffer stroke class in the sequential machines, Mem. Fac. Sci. Kyusyu Univ., Vol. A13, No. 2, 1959.
177. HUZINO,S., Theory of finite automata, Mem. Fac. Sci. Kyusyu Univ., Vol. A16, No. 2, 1962.
178. HUZINO, S., Turing transformation and strong computability of Turing computers, Mem. Fac. Sci. Kyusyu Univ., Vol. A13, No. 2, 1959.
179. HUZINO, S., On some sequential equations, Mem. Fac. Sci. Kyusyu Univ., Vol. A14, No. 1, 1960.
180. KARNAUGH, M., The map method for synthesis of combinatorial logic circuits, AIEE Trans., Vol. 72, 1953.
181. KELLER, H. B., Finite automata, pattern recognition and perceptrons, J. Assoc. Comp. Machines, Vol. 8, No. 1, 1961.
182. LEE, C. I., Automata and finite automata. Bell Syst. Techn. Journal, Vol. 39, No. 5, 1960.
183. LEE, C. I., Categorizing automata by *W*-machine programs, J. Assoc. Comp. Machines, Vol. 8, No. 3, 1961.
184. LOW, P. R., and MALEY, G. A., Flow table logic, Proc. IRE, Vol. 49, No. 1, 1961.
185. McCLUSKEY, E. J., Minimization of Boolean functions, Bell. Syst. Techn. Journ., Vol. 35, 1956.
186. McCLUSKEY, E. J., Minimal sums for Boolean functions having many unspecified fundamental products, Communication and Electronics, No. 63, November, 1962.
187. McNAUGHTON, R. F., and YAMADA, H., Regular expressions and state graphs for automata, IRE Trans., Vol. EC-9, No. 1, 1960.

188. McNAUGHTON, R. F., The theory of automata, a survey. Advances in Computers, Vol. 2, ed. by Franz L. Alt, 1962.
189. MAYEDA, W., Synthesis of switching functions by linear graph theory, IBM Journ. of Research and Develop., Vol. 4, No. 3, 1960.
190. MEALY, G. H., A method of synthesizing sequential circuits, Bell. Syst. Techn. Journ., Vol. 34, Sept., 1955.
191. MEZEI, J. E., Minimal characterizing experiments for finite memory automata, IRE Trans., Vol. EC-10, No. 2, June, 1961.
192. MLEZIVA, M., The theory of finite automata (neutral nets), Rokvoky matematiky, fysiky a astronomie, Vol. 5, No. 6, 1960.
193. MOISIL, G. C., Mathematical logic and modern technology, Acad. RPR, Inst. de Stud. Romino-Sovietia, 1960.
194. MOISIL, G. C., Rapport sur le developpement dans la R. P. R. de la théorie des mécanismes automatiques, Analele Universităţii C. I. Pahon, Bucureşti, Seria Acta Logica, No. 1, 1959.
195. MOISIL, G. C., Behavior of a multistage system with an ideal relay, Acad. RPR, Bucharest, 1960.
196. MOORE, E. F., A simplified universal Turing machine, Proc. Assoc. Comp. Mach., 1953.
197. MOTT, T. H., Jr., An algorithm for determining minimal normal forms of an incomplete truth function, Commun. and Electronics, No. 3, March, 1961.
198. MULLER, D. E., and BARTKY, W. S., A theory of asynchronous circuits, Intern. Symp. on the Theory of Switching. Part I. The Annals of the Comp. Lab. Harvard Univers., Vol. 29, 1959.
199. MYHILL, J., Finite automata and the representation of events, WADC Technical Report, No. 57-624, 1957.
200. MYHILL, J., Linear bounded automata, U. S. Govt. Research Reports, Vol. 35, March 10, 1961.
201. NASLIN, P., Introduction à l'étude des automatismes à séquences. 1-e partie, Les fonctions logiques et les circuits combinatoires, Automatisme, Vol. 3, No. 1, 1958.
202. NASLIN, P., Circuits à relais et automatismes à séquences, Dunod, Paris, 1958.
203. NASLIN, P., A note on the simplification of Boolean functions, Process Control and Automation, Vol. 8, No. 6, 1961.
204. NEDELCU, M., Teoria algebrica a schemelor cu functionare discreta, Automatica şi electronica, Vol. 4, No. 6, 1960.
205. NERODE, A., Linear automaton transformations, Proc. Amer. Math. Society, Vol. 9, 1958.
206. NETHERWOOD, D. B., Minimal sequential machines, IRE Trans., Vol. EC-8, No. 3, 1959.

207. NEUMANN, J. von, The general and logical theory of automata. In coll.: Cerebral Mechanisms in Behaviour, N. Y., 1951.
208. OTT, G., and FEINSTEIN, N. H., Design of sequential machines from their regular expressions, J. Assoc. Comp. Machines, Vol. 8, No. 4, 1961.
209. PAULL, M. C., and UNGER, S. H., Minimizing the number of states in completely specified sequential switching functions, IRE Trans., Vol. EC-8, No. 3, 1959.
210. PERCUS, J. K., Matrix analysis of oriented graphs with irreducible feedback loops, IRE Trans., Vol. CT-2, 1956.
211. PFEIFER, J. E., Symbolic logic, Scientific American, Vol. 183, No. 6, 1950.
212. PHISTER, M., Jr., Logical Design of Digital Computers, John Wiley & Sons, N. Y., 1959.
213. PUTNAM, H., Decidability and essential undecidability, Journ. Symb. Logic, Vol. 22, 1957.
214. QUINE, W. V., Way to simplify truth functions, Amer. Math. Monthly, Vol. 62, 1955.
215. RIEGER, O teorii neuronovych siti, Aplikace Matematiky, Vol. 3, No. 4, 1958.
216. ROTH, J. P., Minimization of nonsingular Boolean trees, IBM Journ. of Research and Develop., Vol. 3, No. 4, 1959.
217. ROTH, J. P., Minimization over Boolean trees, IBM Journ. of Research and Develop., Vol. 4, No. 5, 1960.
218. ROTH, J. P., Algebraic topological methods for the synthesis of switching systems, I. Trans. Amer. Math. Soc., Vol. 88, July, 1958.
219. RUBINOFF, M., Remarks on the design of sequential circuits, International Symp. on the Theory of Switching, Part II. Annals of the Comp. Lab. Harvard University, Vol. 30, 1959.
220. SCHUBERT, E. J., Matrix algebra of sequential logic, Communic. and Electronics, Vol. 46, No. 1, 1960.
221. SCHUBERT, E. J., Simultaneous logical equations, Communic. and Electronics, Vol. 46, No. 1, 1960.
222. SCHUBERT, E. J., Symmetric switching functions, Communic. and Electronics, Vol. 46, No. 1, 1960.
223. SCHÜTZENBERGER, M. P., A remark on finite transducers, Information and Control, Vol. 4, No. 2-3, 1961.
224. SCHÜTZENBERGER, M. P., On the definition of a family of automata, Information and Control, Vol. 4, No. 2-3, 1961.
225. SECHU, S., On electric circuits and switching circuits, IRE Trans., Vol. CT-3, 1956; Vol. CT-4, No. 3, 1957.

226. SECHU, S., Mathematical models for sequential machines, IRE Convent. Record, Vol. 7, part 2, 1959.
227. SECHU, S., MILLER, R. E., and METZE, G., Transition matrices of sequential machines, IRE Trans., Vol. CT-6, No. 1, 1959.
228. SEMON, W., Matrix methods in the theory of switching, International Symposium on the Theory of Switching, Part. II. Annals Comp. Lab. Harvard Univ., Vol. 30, 1959.
229. SHANNON, C. E., Von Neumann's contributions to automata theory, Bull. Amer. Math. Soc., Vol. 64, part 2, 1958.
230. SHANNON, C. E., Computers and automata, Proc. IRE, Vol. 43, 1953.
231. SHANNON, C. E., A symbolic analysis of relay and switching circuits, Trans. AIEE, Vol. 57, 1938.
232. SIMON, J. M., A note on the memory aspect of sequence transducers, IRE Trans., Vol. CT-6, No. 1, 1959.
233. SIMON, J. M., Some aspects of the network analysis of sequence transducers, Journ. Franklin Inst., Vol. 65, June, 1958.
234. SRINIVASAN, C. F., and NARASIMHAN, R., On the synthesis of finite sequential machines, Proc. Indian Acad. Sci., Vol. A50, No. 1, 1959.
235. SRINIVASAN, C. V., State diagram of linear sequential machines, Journ. Franklin Inst., Vol. 273, 1962.
236. SUCHESTON, L., Note on mixing sequences of events, Acta Mathematica, Acad. Sci. Hung., Vol. 11, fasc. 3-4, 1960.
237. SUTO, S., and WATANABE, T., The theory of analysis of sequential time switching circuits, Journ. Inst. Electr. Communic. Engrs. (Japan), Vol. 42, No. 6, 1959.
238. TANG, D. T., Analysis and synthesis techniques of oriented communications nets, IRE Trans., Vol. CT-8, No. 1, 1961.
239. TURING, A. M., On computable numbers with an application to the entscheidungsproblem, Proc. London Math. Soc., Vol. 42, 1936-1937. With a correction, Vol. 43, 1947.
240. UNGER, S. H., Hazards and delays in asynchronous sequential switching circuits, IRE Trans., Vol. CT-6, No. 1, 1959.
241. URBANO, R. H., and MUELLER, R. K., A topological method for the determination of minimal forms of Boolean functions, IRE Trans., Vol. EC-5, No. 3, 1956.
242. WALD, A., Sequential Analysis, John Wiley & Sons, N. Y., 1947.
243. WANG, HAO, Circuit synthesis by solving sequential Boolean equations, Zeitschrift für Mathematische Logik und Grundlagen der Math., Vol. 5, 1959.

244. WANG, HAO, A variant to Turing's theory of computing machines, J. Assoc. Comp. Machines, Vol. 4, Jan., 1961.
245. WATANABE, S., Five-symbol 8-state and 5-symbol 6-state universal Turing machines, *Ibid.*, Vol. 8, No. 4, 1961.
246. WATANABE, S., On a minimum universal Turing machine, MCB Report, Tokyo, 1960.
247. WHITESITT, J. E., Boolean Algebra and Its Applications, Addison-Wesley Pub. Co. Inc., Reading, Mass., 1961.

SUPPLEMENT TO BIBLIOGRAPHY

248. AIZERMAN, M. A. et al., Transformation of rhythm in sequential machines and the synthesis of relay circuits, Avtomatika i Telemekhanika, Vol. 23, No. 11, 1962.
249. BELYAKIN, N. V., A class of Turing machines, Doklady Akad. Nauk SSSR, Vol. 148, No. 1, 1963.
250. BERGE, C., Theory of Graphs and Its Applications, John Wiley & Sons, N. Y., 1962.
251. BURKS, A., and WRIGHT, J., The theory of logical nets, Kiberneticheskiy Sbornik, No. 4, IL, Moscow, 1962.
252. GLUSHKOV, V. M., The Synthesis of Digital Automata, Fizmatgiz Press, Moscow, 1962.
253. FOZMIDIADI, V. A., and CHERNYAVSKY, V. S., The ordering of the set of automata, Voprosy Teorii Matem. Machin (a collection), No. 2, 1962.
254. KORPELEVICH, G. M., The relationship between the concepts of solvability and enumerability in regard to finite automata, Doklady Akad. Nauk SSSR, Vol. 149. No. 5, 1963.
255. LEVENSHTEIN, V. I., The inversion of finite automata, Doklady Akad. Nauk SSSR, Vol. 147, No. 6, 1962.
256. LETICHEVSKIY, A. A., Completeness conditions in the class of Moore's automata, Materialy nauchnykh seminarov po teoret. i prikl. voprosam kibernetiki, Vol. 1, No. 2, 1963.
257. LUNTS, A. G., Finite p-adic automata, Doklady Akad. Nauk SSSR, Vol. 150, No. 4, 1963.
258. MOISIL, G. C., Algebraic Theory of Automatic Mechanisms, Pergamon Press, N. Y., 1964.
259. RABIN, M. O., and SCOTT, D., Finite automata and problems to be solved, Kiberneticheskiy Sbornik, No. 4, IL, Moscow, 1962.
260. SIVINSKI, E., The synthesis of multistage systems with delay elements, Archiwum automatyki i Telemechaniki, Vol. 7, Issue 1-2, 1962.

261. SPIVAK, M. A., A new algorithm for the abstract synthesis of automata, Materialy nauchnykh seminarov po teoret. i prikl. voprosam kibernetiki, Vol. 1, No. 3, 1963.
262. SHEPHERDSON, J. C., Reduction of bilateral automata to unilateral automata, Kiberneticheskiy Sbornik, No. 4, IL, Moscow, 1962.
263. YANOV, Yu. I., Identical transformations of regular expressions, Doklady Akad. Nauk SSSR, Vol. 147, No. 2, 1962.
264. ARTHUR, M. E., Geometric mapping of switching functions, IRE Trans., Vol. EC-10, No. 4, December, 1961.
265. BAR-HILLEL, J., and SHAMIR, E., Finite-stage languages: formal representations and adequacy problems, Bull. Res. Concil Israel, Vol. 8F, 1960.
266. BEATTY, J., and MILLER, R. E., Some theorems for incompletely specified sequential machines with application to state minimization. Switching Circuits Theory and Logical Design (Proc. of the Third Annual AIEE Symposium), September, 1962.
267. BRZOZOWSKI, J. A., and McCLUSKEY, E. J., Signal flow graph techniques for sequential circuit state diagrams, U. S. Government Research Reports, Vol. 37, May 20, 1962.
268. BURKS, A. W., and WRIGHT, J. B., Sequence generators, graphs and formal languages, Inform. and Control, Vol. 5, No. 3, September, 1962.
269. ELGOT, C. C., and RUTLEDGE, J. D., Machine properties preserved under state minimization. Switching Circuits Theory and Logical Design (Proc. of the Third Annual AIEE Symposium), September, 1962.
270. GINSBURG, S., and ROSE, G. F., A comparison of the work done by generalized sequential machines and Turing machines, Trans. Amer. Math. Soc., Vol. 103, No. 3, 1962.
271. GRZEGORCZYK, A., Outline of Mathematical Logic, Warsaw, 1961.
272. HARTMANIS, J., Maximal autonomous clocks of sequential machines, IRE Trans., Vol. EC-11, No. 1, 1962.
273. HARTMANIS, J., Loop-free structure of sequential machines, Information and Control, Vol. 5, No. 1, March, 1962.
274. HARTMANIS, J., The equivalence of sequential machine models, IEEE Trans., Vol. EC-12, No. 1, February, 1963.
275. HARTMANIS, J., and STEARNS, R. E., Some dangers in state reduction of sequential machines, Information and Control, Vol. 5, No. 3, September, 1962.
276. JAMADA, H., Disjunctively linear logic nets, IRE Trans., Vol. EC-11, No. 5, October, 1962.

277. KAUTZ, W. H., Some unsolved problems in switching theory. Switching Circuits Theory and Logical Design, conference paper, Detroit, October, 1962.
278. McCLUSKEY, E. J., Jr., Minimum state sequential circuits for a restricted class of incompletely specified flow tables, Bell Syst. Tech. J., Vol. 41, No. 6, November, 1962.
279. NARASIMHAN, R., Minimizing incompletely specified sequential switching functions, IRE Trans., Vol. EC-10, No. 3, September, 1961.
280. NEMITZ, W., and REEVES, R., A mathematical theory of switching circuits, Mathematical Magazine, Vol. 33, No. 1, 1959.
281. SHEPHERDSON, J. C., and STURGIS, H. E., Computability of recursive functions, J. Assoc. Comp. Machines, Vol. 10, No. 2, April, 1963.
282. WEEG, G. P., Finite automata and connection matrices, Communications of the ACM, Vol. 3, No. 7, 1960.
283. WEEG, G. P., and KATELEY, J., Some properties of strongly connected machines, Communications of the ACM, Vol. 3, No. 7, 1960.
284. ZEMANEK, H., Sequentielle asynchrone logik, Elektronische Rechenanlagen, Vol. 4, No. 6, 1962.

Addenda to Bibliography

1. BABAYEV, O. B. A method of synthesis of many-moment schemes, Izvestiya Vuzov, Priborostroyeniye, Vol. 7, No. 6, 1964.
2. BARZDIN', YA. M. The capacity of a medium and the behavior of automata, Doklady Akad. Nauk, Vol. 160, No. 2, 1965.
3. BODNARCHUK, V. G. Systems of equations in the algebra of events, Zh. Vychislit. Matemat. i Matemat. Fiz., Vol. 3, No. 6, Nov.-Dec., 1963.
4. VAVILOV, Ye. N. and OSINSKIY, L. M. Structural synthesis of automata that operate with supplementary moments (in Ukrainian), Avtomatika, No. 2, 1964.
5. GECHEG, F. On the group of one-to-one transformations defined by finite automata, Kibernetika, No. 1, 1965.
6. GLUSHKOV, V. M. Introduction to Cybernetics, Acad. Sci. Press, Ukr. SSR, Kiev, 1964.
7. GLUSHKOV, V. M. Theory of automata and some applications of it, Vestnik Akad. Nauk SSSR, No. 7, 1964.
8. DOBROV, D. Extreme automata, Fiz.-Matem. Spisaniye, Vol. 7, No. 1, 1964.
9. Completed works in the field of the theory of relay devices, finite automata, and coding (1962-1963), Pribory i Sredstva Avtomatizatsii, No. 10, 1964.
10. ZAKROVSKIY, A. D. On the shortening of sorting in the solution of certain problems in the synthesis of discrete automata, Izvestiya Vuzov, Radiofizika, 1964, Vol. 7, No. 1.
11. ZAROVNYY, V. P. On the group of automatic one-to-one mappings, Doklady Akad. Nauk, Vol. 156, No. 6, 1964.
12. ZAROVNYY, V. P. Automatic interlacing of groups, Doklady Akad. Nauk, Vol. 160, No. 3, 1965.
13. ZYKIN, G. P. A remark on a theorem of Hao Wang, Algebra i Logika (Institute of Mathematics of the Siberian Division of the Academy of Sciences of the USSR), Vol. 2, No. 1, 1963.

14. ION, I. D. On the connection between theorems of algorithms and the abstract theory of automata, Revue de Mathématiques Pures et Appliquées (BPR), Vol. 8, No. 4, 1963.
15. KOLMOGOROV, A. N. Discrete automata and finite automata (Text of an address), Proceedings of the 4th All-Union Mathematical Congress, Vol. 1, plenary addresses, Acad. Sci. Press, Leningrad, 1963.
16. KRATKO, M. I. On the reducibility of a combinatorial problem of Post to certain mass problems in the theory of finite automata, Sbornik Vychislitel'nyye Sistemi, No. 9 Novosibirsk, 1963.
17. KRATKO, M. I. Algorithmic insolvability of a problem in the theory of finite automata, Diskretnyy Analiz, No. 2, Novosibirsk, 1964.
18. KRATKO, M. I. On the existence of nonrecursive bases of finite automata, Algebra i Logika, seminar, Vol. 3, No. 2, 1964.
19. KRATKO, M. I. Algorithmic insolvability of problem of recognition of completeness for finite automata, Doklady Akad. Nauk, Vol. 155, No. 1, 1964.
20. LAZAREV, V. G., and PIYL', Ye. I. On certain classes of finite automata, Zh. Vychislit. Matemat. i Matemat. Fiz., Vol. 2, No. 4, July-August, 1962.
21. V. G. LAZAREV and PIYL', Ye. I. Synthesis of Asynchronic Finite Automata, Nauka Press, 1964.
22. LEVENSHTEYN, V. I. On stable extension of the definition of partial automata, in the collection: Problemy Kibernetiki, No. 10, edited by A. A. Lyapunov, Fizmatgiz Press, Moscow, 1963.
23. LETICHEVSKIY, A. A. On the minimization of finite automata, Kibernetika, No. 1, 1965.
24. LUNTS, A. G. A method of analysis of finite automata, Doklady Akad. Nauk, Vol. 160, No. 4, 1965.
25. LYUBICH, Yu. I. Estimates for an optimal determination of indeterminate autonomous automata, Sibirskiy Matematicheskiy Zhurnal. Vol. 5, No. 2, March-April, 1964.
26. MARKOV, V. L. Turing machines and binary automata, Sibirskiy Matematicheskiy Zhurnal, Vol. 5, No. 1, January-February, 1964.
27. PIYL', Ye. I. A method of distribution of internal states of a finite automation, in the collection: Printsipy Postroyeniya Setey i Sistem Upravleniya," Nauka Press, Moscow, 1964.
28. SORKIN, Yu. I. The theory of determining relations for automata, in the collection: Problemy Kibernetiki, No. 9, Fizmatgiz Press, Moscow, 1963.

29. SRAGOVICH, V. G. and Yu. A. FLEROV. Construction of a class of optimal automata, Doklady Akad. Nauk, Vol. 159, No. 6, 1964.
30. TRAKHTENBROT, B. A. On an estimate of the weight of a finite tree, Sibirskiy Matematicheskiy Zhurnal, Vol. 5. No. 1, January-February, 1964.
31. The Theory of Finite and Probabilistic Automata, Proceedings of the International Symposium on the Theory of Relay Devices and Finite Automata (IFAK), Nauka Press, Moscow, 1965.

Index