The Logic of Automata*—Part I

ARTHUR W. BURKS AND HAO WANG

University of Michigan, Ann Arbor, Mich.

1. INTRODUCTION†

We are concerned in this paper with the use of logical systems and techniques in the analysis of the structure and behavior of automata.

In Section 2 we discuss automata in general. A new kind of automaton is introduced, the growing automaton, of which Turing machines and self-duplicating automata are special cases. Thereafter we limit the discussion to fixed, deterministic automata and define their basic features. We give methods of analyzing these automata in terms of their states. Four kinds of state tables—complete tables, admissibility trees, characterizing tables, and output tables—are used for this purpose. These methods provide a decision procedure for determining whether or not two automaton junctions behave the same. Finally, a class of well-formed automaton nets is defined, and it is shown how to pass from nets to state tables and vice versa. A coded normal form for nets is given.

In Section 3* we show how the information contained in the state tables can be expressed in matrix form. The (i, j) element of a transition matrix gives those inputs which cause state S_i to produce state S_j. Various theorems are proved about these matrices and a corresponding normal form (the decoded normal form or matrix form) for nets is introduced.

In Section 4* we first show how to decompose a net into one or more subnets which contain cycles but which are not themselves interconnected cyclically. We then discuss the relation of cycles in nets to the use of truth functions and quantifiers for describing nets. We conclude by relating nerve nets to other automaton nets.

2. AUTOMATA AND NETS

2.1. *Fixed and Growing Automata*

To begin with we will consider any object or system (e.g., a physical body, a machine, an animal, or a solar system) that changes its state in time; it may or may not change its size in time, and it may or may not interact with its environment. When we describe the state of the object at any arbitrary time, we have in general to take account of: the time under consideration, the past history of

* Received December, 1956. Sections 3 and 4 will appear as part II in the next issue of the Journal. This research was supported by the United States Air Force through the Air Force Office of Scientific Research of the Air Research and Development Command under Contract No. AF 18(603)-72.

† We wish to thank Irving M. Copi, Calvin C. Elgot, John H. Holland, and especially Jesse B. Wright for many helpful suggestions.

the object, the laws governing the inner action of the object or system, the state of the environment (which itself is a system of objects), and the laws governing the interaction of the object and its environment. If we choose to, we may refer to all such objects and systems as automata. The main concern of this paper is with a special class of these automata: viz., digital computers and nerve nets. To define this class as a subclass of automata in general, we will introduce various simplifying and specifying assumptions. It will become clear that in adopting each assumption we are making a deliberate and somewhat arbitrary decision to confine our attention to a certain subclass of automata. For example, by altering some of the decisions we arrive at the rather interesting concept of indefinitely growing automata, which include the well-known Turing machines as particular cases.

A. *Discrete Units.* The first decision we make is to use only discrete descriptions; this means that between any two moments and between any two elements (particles, cells, etc.) there is a finite number of other moments or elements. This decision is a consequence of our interest in digital computers. It carries with it a commitment to emphasize discrete mathematics in the analysis of the systems under investigation: recursive function theory and symbolic logic. Hence our problems differ from the more common ones in which time, the elements of a system, the states (color, hardness, etc.) of an element, and the interaction of an object with its environment are all treated as continuous. When this is done, the emphasis is naturally placed on classical analysis and its applications. In contrast with digital computers, which use discrete units, analog computers simulate in a continuous manner, and for the study of them continuous (nondiscrete) mathematics is especially appropriate.

It should be remarked that, though discrete mathematical systems are generally more useful for the investigation of discrete automata, a very common (and perhaps at the present time even primary) scientific use of digital computers is to represent (approximately, of course) continuous mathematics (e.g., to solve differential equations). In effect, this procedure involves finding discrete mathematical systems which adequately approximate the particular continuous system at hand.

B. *Deterministic Behavior.* We will not deal with elements capable of random (nondeterministic) behavior. Rather, we will assume that at each time the complete state of an object is entirely determined by its past history, including the effects of its environment throughout the past. Statistics could be employed to treat deterministic automata containing large numbers of elements (cf. the kinetic theory of gases), but we will not do this here.

C. *Finitude of Bases.* We will always exclude an "actual infinity" of states: each system will contain a finite number of elements at each time, and each element can be in only one of a finite number of states at any time. We will reserve for an independent decision the possibility of a "potential infinity" of elements and states, i.e., whether or not the number of elements or states of each element may change with time and in particular increase without bound.

The finitude of time requires separate treatment. The discreteness condition

(A) implies that there are never infinitely many moments between two given times. But this still leaves open the question as to how many different times are to be considered. For a general theory it would be inelegant to take a definite finite number, say, 10^{18} or 10^{27}, as an upper bound to the number of possible moments. It seems desirable to allow time to increase indefinitely and to study the behavior of an object through all time. The one remaining choice is the question of an infinite past. The assumption of an infinite past has the advantage of making the entire time sequence homogeneous, thereby destroying a major part of the individuality of each moment of time. However, in the presence of our deterministic assumption an infinite past would be inconvenient (for reasons to be given in Section 3.3), so we stipulate that there is a first moment of time (zero). The internal state of the automaton at time zero will be some distinguished state.

Speaking arithmetically, the two alternatives are to represent moments of time by positive (or nonnegative) integers or by all integers (including the negative ones).

(C1) An infinite future but a finite past: Every nonnegative integer represents a time moment and vice versa; the number zero represents the beginning of time.

(C2) An infinite future and an infinite past.

Our decision is to adopt the first of these two alternatives.

D. *Synchronous Operations.* Some computers contain component circuits which operate at different speeds according to their function, location, the time, or the input at the time. Such circuits are called "asynchronous," in contrast to synchronous circuits which work on a uniform time scale (usually under the direction of a central control clock). There are really two aspects to asynchronous operation: (1) the actual intervals of time between operations vary with the time; (2) different parts of the computer pass through their states at different rates, these rates depending, in general, on the information held in the circuits.

We will assume that all elements of a given system (net) operate at the same rate, and we will call this the "synchronous" mode of operation. In particular, this means an element may operate at each nonnegative integer t. This assumption does not imply that all time intervals are equal; e.g., the interval from time 7 to time 8 can be one microsecond, while the interval from time 8 to time 9 is 12 hours. Thus our assumption does not exclude possibility (1) of the preceding paragraph. It involves some restriction with regard to point (2), but not as much as one might think. For example, we can represent different parts of an asynchronous machine by subsystems operating at different rates, interconnecting these with logical representations of interlocks. And just as discrete systems (e.g., digital computers) can be used to simulate continuous systems [cf. the second paragraph under the discussion of assumption (A)], so synchronous computers can be used to simulate asynchronous ones.

E. *Determination by the Immediate Past and the Present.* The stipulation that the behavior of an automaton at time $t + 1$ is determined by the past leaves unsettled the question as to how the remote past influences the present. We will assume that such influence occurs indirectly through the states at intermediary time moments, so that to calculate the state of the automaton at $t + 1$ it is not necessary to know its state for any time earlier than the preceding moment t. Thus we assume that for the present to be influenced by an event which happened in the remote past a record of that event must have been preserved internally during the intervening time. This postulate corresponds closely to the actual mode of operation of automatic systems. Of course, a computer with, e.g., a microsecond clock, may have delay lines, e.g., 500 microseconds long, so that a stored pulse is not accessible to the arithmetic unit at every clock time, but such a delay line is naturally represented as a chain of 500 unit (microsecond) delays, with the input and output of the chain connected to the rest of the computer. Indeed, the assumption we are now making causes no significant loss in generality, because for each fixed N such that an automaton A can always remember what happened during the N immediately preceding moments (but not more), we can easily devise an automaton A' which, though capable of directly remembering only the immediate past, simulates completely the automaton A.

When we think of automata which have unbounded memory or, in particular, ones which can remember everything that has happened in the past, we encounter, a basically different general situation. In such cases, the information to be retained increases with time, whence for any automaton of fixed capacity there may come a time beyond which it can no longer hold all the information accumulated in its life history. Thus, for a machine to remember all past history it is necessary and sufficient that it grow in some suitable fashion. Such growth can be accomplished by the appearance at each moment of a new delay element for each automaton input. In short, in the presence of the postulate of determination by the immediate past, the alternative of remembering all past history is best studied in connection with growing automata.

Another problem connected with determination by the immediate past is the role of present inputs in determining the outputs. It would seem natural to stipulate that the environment at time $t + 1$ cannot influence the outputs of the automaton at time $t + 1$ but only at time $t + 2$. That is the case with neural nets, since each neuron has a delay built into it. Strictly speaking, it is true in computer nets, but because the delay in a switch may be small compared to the unit delay of the system, it is convenient to regard this switching action as instantaneous. Thus the well-formed nets of Burks and Wright [1], which will be discussed further in Section 2.3, permit outputs which are switching functions of the inputs.

F. *Automata and Environment.* Supposedly the change of state of a solipsist is independent of the environment, and the environment is not affected by the solipsist; cf. Leibnitz's concept of a "monad." A "solipsistic" automaton would be one which (1) changed its state independently of any environmental changes and (2) whose output did not influence the environment. We are not primarily interested in such automata. Rather, we will consider how the environment (the inputs) affects the automaton but not how the output of an automaton affects its environment.

The last point is related to the ordinary method of representing inputs and outputs. It is well known that there are significant and useful logical symbols for the internal action of automata (see Section 2.3). The standard method of representing inputs and outputs is in terms of the binary states of input and output wires. This is not directly applicable in simulating such "inputs" as light and sound waves, physical pressures, etc., and such "ouputs" as physical actions. Theoretically, we can, however, just as well interpret certain standard binary elements as representing these. For some purposes, we may want to add as new primitives representations of the lights and keys commonly used on computers (see Burks and Copi [2], p. 306), as well as symbols representing additional methods of sensing and other methods of acting on the environment that automata are capable of. Von Neumann has done some work along this line, but it has not been published (see Shannon [3], p. 1240). It might be suggested that one ought to devise a symbolism for magnetic or paper tape input and output to a computer; but that is unnecessary because such devices are very well represented by net diagrams for a serial type of storage (see Burks and Copi [2], p. 313, ftn. 9).

We will not here attempt to devise separate notations for the various kinds of interactions possible between an automaton and its environment, but will content ourselves with the customary way of simulating inputs and outputs by binary states of wires. Even subject to this restriction there are a number of alternatives to consider. The most general case would be to identify the environment partly or wholly with certain automata so that interaction occurs among these and the particular automaton under study (cf. the many-body problem of mechanics). A simple case would be to identify the whole environment with another automaton (cf. the two-body problem of mechanics). Accordingly, we have the following alternatives.

(F0) An object changes its state automatically, independent of the environment.

(F1) An automaton changes its state in accordance with its structure and the inputs (the environment).

(F2) Different automata interact with one another.

We will be primarily concerned with (F1). In other words, we will assume that the automaton has no influence over what inputs it receives, and that in general the inputs do have effects on the internal state (i.e., state of internal cells) of the automaton. As a consequence, we can define the units or atoms of which an automaton is compounded into two classes: input cells and internal cells, or input and internal wires, or input and internal junctions.

The situation under (F0) becomes a special case of that under (F1) when either the number of input cells (or wires) is zero (a limiting case) or the effects of the inputs are more or less canceled so that the automaton behaves in an input-independent manner. The latter case is exemplified by a logical element whose output wire is always active regardless of the state of the input wire.

(F2) may also be regarded as a special case of (F1). Since the inputs and outputs of an automaton are wires, two automata may be interconnected to produce a single (more complex) automaton, of which the original automata are parts or subsystems. Thus we regard (F2) as a special case of analyzing a complex machine into interrelated submachines. A common application of this concept is to be found in the design of a general-purpose computer. Typically such a computer is divided into Arithmetic Unit, Storage, Input-Output Unit, and Control (see, for example, Burks and Copi [2], p. 301). The utility of making such divisions lies partly in the relatively independent functioning of these units and partly in the (related) fact that it is conceptually easier to understand what goes on in terms of these parts. The kind of structuring under discussion usually occurs at more than one level; e.g., the Parallel Storage (of Burks and Copi [2], pp. 307–313) divides naturally into a switch and 4096 bins (each storing a word), and the bins are in turn "composed" of cells (each storing one bit of a word).

G. *Exclusion of Growth.* While we have adopted the postulate of finite bases, we have yet to decide whether the structure of an automaton, the number of its cells, or the number of possible states of each cell are to be allowed to change with time. If changes are permitted but are confined by a preassigned finite bound, we might as well have used a fixed automaton which embodies this bound to begin with. Hence the really interesting new case is that of a growing automaton which has no preassigned finite upper bound on the possible number of cells or cell states. Structural changes (e.g., rewiring a given circuit) do not seem to generate unbounded possibilities, although in special studies, such as investigations into the mode of operation of the human brain (cf. Rochester *et al.* [4]), the use of a structurally changing automaton is more illuminating than the use of the corresponding fixed automaton.

In any case, we can, theoretically, reduce all three kinds of growth to increase in either the number of cells or the number of possible cell states: given any growing automaton, we can find another which functions in the same way but grows only in the number of its cells (or, alternatively, only in the number of possible states for each cell). For every and all forms of growth, it seems natural (in the context of our deterministic assumption) to require that the process be effective (recursive). We will therefore assume once and for all that each definition of a growing automaton determines an effective method by which we can, for each time t, construct the automaton and determine its state for that time. An important particular case corresponds to primitive recursive definitions, each of which yields a method by which we can construct the automaton for $t = 0$ and, given the automaton *and its state* at t, we can construct the automaton at $t + 1$. The growth may not depend on the state or the inputs, but the possibility of its doing so is provided for. Moreover, "growth" is taken to include shrinkage as well as expansion. Thus we could have a "growing" computer which expands and contracts as the computation proceeds, having at each time period just the capacity needed to store the information existing at that time.

Two types of automata, *fixed* and *growing*, can be characterized as follows:

(G1) The structure and cells of the automaton are fixed once and for all, and each cell is capable of a fixed number of states.

(G2) The automaton may grow (expand *and* contract) in time in a predetermined effective manner.

In this paper we will be concerned entirely with fixed automata, except for some remarks on growing nets in this subsection. These remarks are intended to elucidate the concept of a growing net and to indicate why we think it is important. But before beginning on them we wish to specify (G1) further by stipulating that each cell, junction, or wire is capable of two states, on and off, firing and quiet; we will later correlate these with one and zero, and with true and false. We could of course allow each cell to have any fixed finite number of possible states and different cells to have different numbers of states. But it is better to fix the number of states at the constant two. There are a number of reasons for this. The wires and cells of many automata and most digital computers do in fact have two significant states. When this is not the case we can always represent a cell with q possible states by p two-state cells for any $p \geqq \log_2 q$ (e.g., ten of the sixteen different states of four binary net wires can represent ten discrete electrical states of a single circuit wire), so by adapting our system to the commonest case we do not lose the power to treat the nonbinary cases. This commitment to two-valued logic need not blind us to the fact that there may be cases where multivalued logic is more convenient; the point is that our logic can handle these cases and we have no interest at the moment in exploiting whatever advantages multivalued logic might have here.

We return now to growing nets, mentioning first some special cases of them already known. A Turing machine (see Turing [5]; Kleene [6]; Wang [7, 8]) may be regarded as an automaton with a growing tape. Usually the tape is regarded as infinite, but at any time only a finite amount of information has been stored on it, so it is essentially a finite but expanding automaton net (cf. Burks and Copi [2], p. 313, ftn. 8). If, in a Turing machine, we take as input cells the squares included on the minimum consecutive tape position which contains all marked squares at the moment, then the growth consists simply of the expansion and contraction of the tape. Or if we use the formulation of Wang [7] which eliminates the erasure operation, a Turing machine is a growing automaton with an even more limited type of growth—namely, an expansion of the tape. In contrast, a growing automaton may in general grow anywhere, not only at the periphery but also internally (by having new elements arise between elements already present).

Though a Turing machine is a special kind of growing automaton, it has as much mathematical (calculating) ability as any growing automaton; for every type of computation can be done by some Turing machine, and the mathematical ability of an automaton is limited to computation. In view of this situation one might wonder why the general concept of a growing net is of interest. Its importance can be shown by the following considerations.

John von Neumann has developed some models of self-reproducing machines (von Neumann [9]; Shannon [3], p. 1240; Kemeny [10], pp. 64–67). These are machines that grow until there are two machines, connected together,—the original one and a duplicate of it; the two machines may then separate. Hence they are clearly cases of growing nets.

The basic process to be simulated or modeled in the growth and reproduction of living organisms is the complete process from a fertilized egg to a developed organism which can produce a fertilized egg. For this purpose we would need to design a relatively small and simple automaton which would grow to maturity (given an appropriate environment) and would then produce as an offspring a new small automaton. Von Neumann's models can be construed either at the level of cells or at the level of complete organisms, but in either case they seem to provide only a partial solution. The process of cell duplication is only one component of the complete process described above, and the self-reproduction of a completely developed entity omits the important process of development from infancy to maturity. Hence the model we suggest is a type of growing automaton not yet covered in the literature.

A second novel type of growing automaton is a generalized Turing machine in which growth is permitted at points other than at the ends of the tape. A typical Turing machine, although logically powerful, is clumsy and slow in its operation. Consequently, to design a special-purpose Turing machine or to code a program for a universal Turing machine is a complicated and laborious process (although a completely straightforward one). What complicates the task is the linear arrangement of information on a single tape, which requires the tape or reading-head to be moved back and forth to find the information. That movement may be reduced somewhat by shifting the old information around to make room for the new information, but this operation also contributes to the complexity of the whole process. To develop this point further, we will discuss in more detail the relation between recursive functions and Turing machines. Turing [5] worked in terms of computable numbers; Kleene [6] and Wang [7, 8] work in terms of recursive functions. Since our discussion has been in terms of functions, we will use Kleene's and Wang's works as our references.

The basic mathematical result underlying the significance of Turing machines is the following. Mathematicians have rigorously characterized a set of functions, called partial recursive functions, and this set of functions is in some sense equivalent to what is computable (Kleene [6], Ch. XII). Each partial recursive function is definable by a finite sequence of definitions, each definition being of one of six possible forms (Kleene [6], pp. 219 and 279). It is known how to translate each sequence of definitions into a special-purpose Turing machine and into a program for a general-purpose Turing machine (Kleene [6], Ch. XIII; Wang [7]). This translation, while rigorous and straightforward, is often complicated for the reasons, among others, mentioned in the preceding paragraph. Simpler and more direct translations can be made by using growing nets in which growth is allowed to occur whenever it simplifies the construction, not just at two places, i.e., at the ends of the tape, where it is allowed to occur in the conventional Turing machine. Such growing nets will be generalizations of a Turing machine.

We can arrive at a third novel kind of growing automaton by generalizing a general-purpose computer in the way we generalized a Turing machine in the last paragraph. The usual general-purpose computer consists of a fixed internal computer together with one or more tapes. As in the Turing machine, these tapes may be regarded as expanding at the ends whenever needed; in practice the expansion is handled by an operator replacing tape reels, using either blank tape or tape reels from a library of tapes. In writing programs for such a machine, the programmer needs to keep track of two things:

(1) the development of the computation, in terms of the growth of old blocks of information and the appearance of new blocks of information;
(2) shifting the information from one kind of storage to another (e.g., from a serial to a parallel storage) and moving the information about within a storage unit.

Both of these components of computation are essential. But (1) seems more basic for understanding the nature of the computation, and at any rate it is helpful to be able to study each of the components in isolation. This can be done with growing nets, for we can eliminate (2) by providing for growth wherever it is needed to accommodate new information or new connections to old information. We feel that the study of growing automata would contribute to the theory of automatic programming. The development of a powerful *theory* of automatic programming has so far been impeded by the many details involved in actual computation; by eliminating (2) we would eliminate many of these details and would focus attention on the more basic component (1).

We turn now to fixed automata which satisfy the assumptions (A), (B), (C1), (D), (E), (F1), and (G1). In summary, we arrive at the following definition of a (finite) automaton:

DEFINITION 1: A (finite) *automaton* is a fixed finite structure with a fixed finite number of input junctions and a fixed finite number of internal junctions such that (1) each junction is capable of two states, (2) the states of the input junctions at every moment are arbitrary, (3) the states of the internal junctions at time zero are distinguished,[1] (4) the internal state (i.e., the states of the internal junctions) at the time $t + 1$ is completely determined by the states of all junctions at time t and the input junctions at time $t + 1$, according to an arbitrary pre-assigned law (which is embodied in the given structure). An *abstract automaton* is obtained from an automaton by allowing an arbitrary initial internal state.

Several aspects of this definition call for comment. In it automata states have been defined in terms of junction states. This follows Burks and Wright [1], where each wire has the state of the junction to which it is attached, and the nuclei or cell bodies are not regarded as having states but as realizing transformations between junctions or wires. An alternative would be to define automata states in terms of cell states. Condition (4) places some restrictions on the way automata elements are to be interconnected, but it does not completely specify the situation; this will be discussed further in Section 2.3.

The initial state of the internal junctions also calls for discussion. In the definition of an abstract automaton this is taken more or less as an additional input which can be changed arbitrarily. As a result, two abstract automata, to be equivalent, must behave the same for each initial state picked for the pair

[1] This condition applies only to those junctions whose state at a given time does not depend on the inputs at the same time; cf. condition (4) following.

of them. On the other hand, for most applications to actual automata, it is best to assume a single initial state.

The word "structure" in the above definition can be avoided if we speak exclusively in mathematical terms and consider the transformations realized by automata and abstract automata. We will do so in the next subsection, returning to a more detailed investigation of the structure of automata in the following subsection (viz., 2.3).

2.2. *Characterizing Tables and a Decision Procedure*

Consider for a moment automata whose internal states are determined only by the immediate past and hence are not influenced by the present inputs. Let there be M possible input states, $I_0, I_1, \cdots, I_{M-1}$, and N possible internal states, $S_0, S_1, \cdots, S_{N-1}$. Even though each junction of an automaton is capable of only two states, we do not require M and N to be powers of two. For one thing, when an automaton is being defined, the values of M and N are stipulated and are not necessarily powers of two. Also, when an automaton is given, not all possible combinations of internal junction states may occur because of the structure of the automaton, and not all possible combinations of input junctions may be of interest (because, e.g., the automaton is to be embedded in a larger automaton where not all of the possible inputs will be used).

We will assign nonnegative integers to the input and internal states. Let I and S range over these numbers, respectively. A complete automaton state is represented by the ordered pair $<I, S>$. (If the automaton has no inputs, then there are no I's and the complete automaton state is just S.) Let S_0 be the integer assigned to the distinguished initial internal state; S_0 will usually be zero, but not always. An abstract automaton differs from a nonabstract one just in not having a distinguished initial state.

Since the input states are represented by numbers, a complete history of the inputs is a numerical function from the nonnegative integers 0, 1, 2, $\cdots$ (representing discrete times) to integers of the set $\{I\}$. That is, it is an infinite sequence $I(0), I(1), I(2), \cdots, I(t), \cdots$; it may be viewed as representing the real number $\{I(0) + [I(1)/K] + [I(2)/K^2] + \cdots\}$, in which K is the maximum of the set $\{I\}$. By our convention that the initial internal state is S_0 we have $S(0) = S_0$. By the assumption of complete determination by the immediate past we have for all t

$$S(t + 1) = \tau[I(t), S(t)],$$

where τ is an arbitrary function from the integer pairs $\{<I, S>\}$ to the integers $\{S\}$. Or, in other words, as the input function I and the time t are the independent variables, τ is an arbitrary function of two arguments (one ranging over functions of integers and another ranging over integers) whose values are integers. It follows by a simple induction that for each infinite sequence $I(0), I(1), \cdots, I(t), \cdots$, repeated application of the function τ yields a unique infinite sequence $S(0), S(1), \cdots, S(t), \cdots$, with $S(0) = S_0$. Since for many purposes we are interested not only in the existence of values of the function τ, but also in finding them, we will assume that τ is defined effectively, though actually much of our discussion would be valid without this restriction.

We next broaden our theory so as to include automata whose internal state at $t + 1$ depends also on the inputs at $t + 1$. To do this we allow P "output" states $O_0, O_1, \cdots, O_{P-1}$ such that

$$O(t) = \lambda[I(t), S(t)],$$

where λ is again an arbitrary effective function. In general, the complete state of an automaton at any time is given by the ordered triad $<I, S, O>$. In specific cases I, S, or O may be missing.

We can now give an analytic definition of automata and abstract automata by means of these transformations.

DEFINITION 2: An automaton is in general characterized by two arbitrary effective transformations (τ and λ) from pairs of integers to integers. These integers are drawn from finite sets $\{I\}$, $\{S\}$, and $\{O\}$. $\{S\}$ contains a distinguished integer S_0. The transformations are given by

$$S(0) = S_0$$
$$S(t + 1) = \tau[I(t), S(t)]$$
$$O(t) = \lambda[I(t), S(t)].$$

If we omit the condition $S(0) = S_0$, we obtain an abstract automaton.

Thus, speaking analytically, the study of finite automata is essentially an investigation of the rather simple class of transformations τ and λ in the above definition. The definition of the class as thus given is superficially very general in allowing τ and λ to be arbitrary calculable functions. On account of the very restricted range and domain of these functions, however, that generality is only apparent. We can find a simple representation of the class of automaton transformations in the following way.

Since τ is effective, and since its domain and range are finite, we can effectively find for each pair $<I, S>$ the value of $\tau[I(t), S(t)]$. Hence we can produce a table of $M \times N$ pairs $\ll I, S>, S'>$ such that if $<I, S>$ is part of the state at t, then S' is part of the state at $t + 1$. We shall call this set the *M–N complete table* of the given automaton. Each complete table is a definition of the function τ.

In a similar way we can construct an *output table* for the automaton, each row being of the form $\ll I, S>, O>$. Such a table defines the function λ.

It is important that the function τ and the complete table involve a time shift, while the function λ and the output table do not. Hence for an investigation of the behavior of an automaton through time the complete table is basic, the output table derivative. That is, by means of the complete table we can compute $S(1)$, $S(2)$, $S(3)$, $\cdots$ from the inputs $I(0)$, $I(1)$, $I(2)$, $\cdots$ and leave the determination of $O(0)$, $O(1)$, and $O(2)$, $\cdots$ for later. Note that to stipulate that the output at time t cannot be influenced by the input at the same time is to require λ to be such that $O(t) = \lambda[S(t)]$. When this is the case the states $O(t)$ and the output table can be dispensed with since $O(t)$ is completely dependent on $S(t)$. (When the behavior of individual junctions is being investigated, the output table may nevertheless be convenient.) For these various reasons the state numbers $\{I\}$ and $\{S\}$ are more basic than the state numbers $\{O\}$, and the complete table is much more important than the output table. This being so, we will often concentrate on automata whose internal states are determined only by the immediate past and ignore the output table.

Since the states S at t are so defined that they do not depend on the inputs at t, any I can occur with any S, and hence there are $M \times N$ possible pairs $<I, S>$ in the complete table. There are N possible values of S'. Hence there are $N^{M \times N}$ possible complete tables for an M-N automaton.

We will say that two abstract M-N automata, in whatever language they may be described, are equivalent, just in case they have the same complete table. That this definition is proper can be seen from the following considerations. If the complete tables are the same, then for the same initial internal state and the same input functions, the initial complete states in the two automata are the same, and the same complete state at any moment plus the same input functions always yield the same complete state at the next moment. On the other hand, if two complete tables are different, there must be a pair $\ll I, S>, S'>$ in one table but not in both, such that by choosing suitable input functions and a suitable internal state represented by $<I, S>$, we can have the complete state represented by $<I, S>$ realized at time zero, and yet the complete state at time one will have to differ. Since we can find effectively the complete table of a given automaton (the details of this process will be explained in the next subsection) and compare effectively whether two complete tables are the same, we have a decision procedure for deciding of two given abstract M-N automata whether or not they are equivalent.

The situation is more complex with automata which have predetermined initial internal states, for two M-N automata with different complete tables may yet behave the same (be equivalent). This is possible because it can happen that for every pair $\ll I, S>, S'>$ which occurs in one table but not in the other, we can never arrive at the internal state S from the distinguished initial internal state S_0, and hence can never have the complete state $<I, S>$, no matter how we choose the input functions. In such a case, two M-N automata with the same prechosen initial internal state may behave the same under all input functions, despite the fact that they have different complete tables. Hence, identity of complete tables is a sufficient but not necessary condition for equivalence of two automata. To secure a necessary and sufficient condition, it suffices to determine all the internal states which the given initial internal state can yield when combined with arbitrary input words, and then to repeat this process with the internal states thus found, etc. If the two complete tables coincide insofar as all the pairs occurring in these determinations are concerned, the two automata are equivalent. To establish that such determinations will always terminate in a finite time requires an argument: Since there are only finitely many pairs in each complete table, the process of determination will repeat itself in a finite time.

To describe the procedure exactly, we introduce a few auxiliary concepts. We can think of a tree with the chosen initial internal state S_0 as the root. From the root M branches are grown, one for each possible input word I_i with the corresponding internal state at the next moment S_{0i} at the end. These M branches can be represented by $\ll I_0, S_0>, S_{00}>, \cdots, \ll I_{M-1}, S_0>, S_{0,M-1}>$, which all belong to the complete table. If all the numbers $S_{00}, \cdots, S_{0,M-1}$ are the same as S_0, the tree stops its growth. If not, M branches are grown on each S_{0i} such that $S_{0i} \neq S_0$, and such that S_{0i} does not equal any S_{0p} for which M branches have already been grown, and we arrive at $S_{0i0}, \cdots, S_{0,i,M-1}$. If all the numbers S_{0ij} (i, j arbitrary) already occur among $S_0, S_{01}, \cdots, S_{0,M-1}$, then the tree stops its growth. If not, then M branches are grown on each S_{0ij} such that it does not equal S_0, or any of the S_{0p}, or any S_{0pq} for which M branches have already been grown. That is, whenever in the construction we come to an S, if it is already on the tree we stop, else we grow M branches on it, one for each I. This process is continued as long as some new internal state is introduced at every height. Since there are altogether only M *a priori* possible internal states, the height (i.e., the number of distinct branch-levels) of the tree cannot exceed M. For any M-N automaton, we can construct such a tree which will be called the *admissibility tree* of the automaton. We can, of course, start with any state S as the assumed initial state, and this gives us an *admissibility tree relative to* S for an abstract automaton.

Those values of S (including S_0) which appear in the admissibility tree are called *admissible* internal states. All other values of S are inadmissible. (Cf. Burks and Wright [1], pp. 1364, 1359.

If we collect together the ordered pairs which represent branches of the admissibility tree, we obtain a (proper or improper) subset of the complete table, which we shall call the *M-N characterizing table* of the automaton. (As in the case of the admissibility tree, it is easy to define the concept of a *characterizing table relative to* S for an abstract automaton.) In order that two M-N automata be equivalent (i.e., behave the same), it is necessary and sufficient that they have the same characterizing table. Since there is an effective method of deciding whether two M-N automata have the same characterizing table, we have a decision procedure for testing whether two M-N automata are equivalent.

Quite often we are not interested in the whole automaton, but rather in the transformations which particular cells (junctions, wires) of an automaton realize. To discuss this aspect of the situation we need to correlate states of automata with states of the elements of automata. We will do this in two stages; first, by putting state numbers in binary form (in the present subsection), and, next by correlating zero and one with junction states (in the next subsection).

The state numbers I and S are nonnegative integers. The binary representation of states is made simply by putting each state number in binary form, making all the I the same length, and making all the S the same length (by adding vacuous zeros at the beginning when necessary). Let m, n be the number of bits of I, S, respectively, in a characterizing table or complete table in binary form. Clearly m is the least integer as large as (or larger than) the logarithm (to the base two) of the maximum I; similarly with n. Let the bits of I be called $A_0, A_1, \cdots, A_{m-1}$, so that $I = A_0 \frown A_1 \frown \cdots \frown A_{m-1}$ where the arch signifies concatenation. Similarly, let the bits of S be called $B_0, B_1, \cdots, B_{n-1}$, so that $S = B_0 \frown B_1 \frown \cdots \frown B_{n-1}$. In the next subsection we will associate the A's with input junctions and the B's with internal junctions. We speak of the characterizing table in binary form as an *m-n characterizing table.*

It follows from our discussion of characterizing tables that the function τ of definition 2, given by

$$S(0) = S_0$$

$$S(t + 1) = \tau[I(t), S(t)],$$

is a rather simple primitive recursive function (with a finite domain) and that the function $S(t)$ is defined primitive recursively relative to the input function $I(t)$. If our interest is in the transformation realized by a particular internal junction, we use another primitive recursive function σ_i such that $\sigma_i(n)$ gives the ith binary digit of $n (i = 0, 1, \cdots)$. Hence each such junction realizes a transformation $\sigma_i[S(t)]$ or $B_i(t)$ which is primitive recursive relative to $I(t)$ (Burks and Wright [1], Theorem XIV; Kleene [11], Theorem 8).

Since the magnitudes of m and n affect the number of junctions of the corresponding automaton, it is of interest to obtain a minimal representation in terms of bits. Given a characterizing table, one can so rewrite the state numbers as to minimize m and n. That is accomplished by so assigning the numbers that the largest I is smaller than the least power of 2 greater than or equal to M, and similarly for S and N. A special case occurs when the states $I_0, I_1, \cdots, I_{M-1}$ are assigned the numbers $0, 1, \cdots, M - 1$, respectively, and the states S_0, $S_1, \cdots, S_{N-1}$ are assigned the numbers $0, 1, \cdots, N - 1$, respectively (note that the distinguished state S_0 is assigned the number zero). A characterizing table put in this form is said to be in *coded normal form.* Automata nets corresponding to this form will be discussed in the next subsection. (Note that minimizing a complete table does not suffice here, because the number of inadmissible states may be such as to require more bits for representing the set of states than for representing the set of admissible states.)

Another special type of automaton is the decoded normal form automaton; it is of interest in connection with the application of matrices to the analysis of nets. In a *decoded normal form* characterizing table the input words are coded as for the coded normal form, but the internal states $S_0, S_1, \cdots, S_{N-1}$ are assigned the numbers $2^0, 2^1, \cdots, 2^{N-1}$; here an N bit word is needed to represent the N internal states. For six internal states we would have the numbers 100000, 010000, 001000, 000100, 000010, 000001; notice that S_0 has a one on the extreme left, i.e., for S_0, B_0 is one and all other B_i are zero. Automata nets corresponding to decoded normal form characterizing tables will be presented in Section 3.

Each of the A's and B's (bit positions of the binary representations of I and S) is a binary variable. Hence the complete table and, more importantly, the characterizing table are (when put in binary form) kinds of truth tables. Thus we have to large extent reduced the problem of automata description and analysis to the theory of truth functions. Of course the S' in $\ll I, S>, S'>$ is the state at $t + 1$, while S is the state at t, so we need to distinguish different times here and hence to use propositional functions (see Section 4). Nevertheless, as the table shows, we need only a very special form of the theory of propositional functions, in which each time step is a matter of the theory of truth functions. So great is the advantage of this partial reduction to the theory of truth-function logic that we will hereafter assume that all characterizing tables are in binary form. Consequently, we may henceforth use any of the techniques of the theory of truth functions which are applicable, not merely the (often cumbersome) truth-table technique.

We return now to the transformations realized by individual elements of the automata, which involves considering the bits of S, i.e., the B's. In the next subsection each B will be associated with an internal junction, so the analysis is also in terms of junctions. The basic problem is to compare the behavior of two bits or junctions, which may or may not belong to the same automaton or characterizing table.

If the two junctions to be compared belong to the same automaton, then they realize the same transformation (behave the same) if and only if the corresponding bits in the S (or S') entries of the characterizing table are everywhere the same. (The state S_0 need not appear in the S' column of $\ll I, S>, S'>$; every other state which is in the S column is in the S' column and vice versa. Of course, all bits are the same in S_0.) This is so because the values of S are the admissible states of the automaton, and at each moment the internal junctions of the automaton are in just one of these states. Hence the question as to whether two junctions of an automaton behave the same can be decided effectively.

If the two junctions are in two different automata, then it is in general not necessary that the automata have the same number of junctions, i.e., that the characterizing tables have the same number of columns, for them to behave the same. Since the transformations depend ultimately only on the time and the inputs, the number of internal junctions need not be the same; since the behavior of an internal junction may be independent of some inputs, even the number of input junctions may be different. Suppose the two junctions belong to an $m_1 - n_1$ and an $m_2 - n_2$ automaton. Then a necessary and sufficient condition for these junctions to realize the same transformation is that there exist some new $m_3 - n_3$ automaton, with $n_3 = n_1 + n_2$, $m_1 + m_2 \geqq m_3 \geqq \max(m_1, m_2)$, which is obtained from the two given automata by connecting a subset of the inputs of one to a subset of inputs of the other in a one-one fashion, and in which the internal junctions under consideration realize the same transformation. That supplies an effective procedure because there is only a finite number of inputs to each automaton and hence only a finite number of ways to interconnect them, and for each way the question of equivalent behavior can be decided effectively. When the process is conducted on the characterizing tables it involves identifying certain of the columns of the I part of the tables.

It is allowed that the subset of inputs which are interconnected may be null, in which case $m_3 = m_1 + m_2$ and the resulting automaton is just the result of juxtaposing the two original automata. For just as the behavior of a junction or cell may be independent of one of the inputs, so it may be independent of all of the inputs. In this case the junction changes from one state at t to another at $t + 1$ in a uniform manner independent of the states of the inputs at t. In other words, it realizes a transformation which is independent of the input functions; we will call such a transformation an *input-independent transformation* (it was called a "constant transformation" in Burks and Wright [1], p. 1358) and speak of the junction as an *input-independent junction.* The number of internal states of an automaton is finite, and an automaton is completely determined by the immediate past, hence all input-independent transformations must be periodic (Burks and Wright [1], Theorem I). Therefore no automaton can realize the simple primitive recursive input-independent transformation which has the value one if and only if t is a square $(0, 1, 4, 9, \cdots)$ (Burks and Wright [1], Theorem II; Kleene [11], Section 13).

A very special type of automaton is one whose internal junctions are all input-independent junctions. In such a net, which we call an *input-independent net,* there may be input junctions, but these cannot influence the internal state at any time. For each such automaton, complete and characterizing tables can be found which have no input states.

The admissibility tree provides an effective means for deciding whether the behavior of an internal junction or cell is independent of a specified input and hence for deciding whether the behavior of a junction is independent of all inputs (i.e., realizes an input-independent transformation). For this purpose it is helpful to identify all occurrences of a given state on the admissibility tree. Then one can trace the behavior of the automaton by proceeding in cycles around the tree. We will not describe the procedure in detail, but will make a few comments about it. By a direct inspection of the characterizing table we can tell whether a change in an input junction A at t can make a difference in B at $t + 1$. Repeating this process we can find all the junctions that A can influence directly, all that these can influence directly, etc. Since the net is finite, this process will terminate. That is, because of the finite nature of the net there is an interval of time q such that if A can influence the behavior of B, it can do it within the time interval q; this interval may be determined from the structure of the net.

If no input junction influences B_j, then B_j realizes an input-independent transformation, which has been already stated to be periodic. This special case of input-independence can be discovered directly from the characterizing table, for a junction B_j realizes an input-independent transformation if and only if for each S there is a unique value of B_j in S', no matter what I is. The behavior of the input-independent transformation during its initial phase and during its main period can be found from the admissibility tree.

(The problem of deciding whether or not two junctions B_j and B_k realize the same transformation is really a special case of the problem of deciding whether a junction realizes a particular input-independent transformation; for we can have B_j and B_k drive an equivalence element, whose output will be the simple input-independent transformation 11111 $\cdots$ if and only if B_j and B_k realize the same transformation. See the next subsection.)

In our discussion we have for some time ignored the output table. It too can be put in binary form, and since both the O and S entries refer to the same time, the result is a straight truth table (in contrast to the characterizing table, where some columns refer to time t and some to $t + 1$). Hence the preceding results are easily extended to include the case of output tables.

We have not yet considered methods of minimizing the labor required to calculate the admissibility tree and the characterizing table. In many cases it is convenient to work with the equations describing a net by means of variables (see the next subsection) rather than with the values of these variables. In some cases one can go directly from such equations to the characterizing table. It is also possible to decompose many nets so as to reduce greatly the number of states to be considered (see Section 4.1). Other methods of simplifying the work will occur to one who is engaged in it and to one familiar with the methods for simplifying truth-table computation.

Before proceeding further let us briefly summarize the concepts introduced in this subsection.

DEFINITION 3. An automaton is in general characterized by state numbers I, S and O. The complete table of an automaton is the set of all pairs $\ll I, S>, S'>$ such that, for given $I(t)$ and $S(t)$, S' is the value of $S(t + 1)$. The characterizing table of an automaton is the subset of its complete table such that each S and S' in it is admissible. A state S is admissible if and only if it is the distinguished initial state S_0 or it can be arrived at from the initial state by choosing a suitable finite sequence of inputs. An admissibility tree is a graph used in computing the admissible states, beginning with S_0 and proceeding systematically. An output table is a table of pairs $\ll I, S>, O>$, stating a value $O(t)$ for given values of $I(t)$ and $S(t)$. An input-independent junction realizes a transformation whose values are independent of the inputs.

We will conclude this subsection by commenting on the relation of the decision procedure described above for testing whether or not two junctions realize the same transformation to other decision procedures. Recently a decision procedure for Church's formulation of computer logic has been announced.[2] We are not acquainted with this decision procedure and hence cannot compare it with ours. However, we can prove the equivalence of Church's system[3] to ours, from which it follows that the two decision procedures accomplish the same result. We do this in two steps. First, the definition of automata given in Section 2.1 is in all essential respects equivalent to that of a well-formed net in Burks and Wright [1] (this will be shown in the next subsection). Church's simultaneous recursion is a slight generalization of the second definition of determinism given

[2] Joyce Friedman, "Some results in Church's restricted recursive arithmetic," *The Journal of Symbolic Logic, 21*, 219 (June, 1956); this is an abstract of a paper presented at a meeting of the Association for Symbolic Logic on December 29, 1955.

[3] Alonzo Church, Review of Edmund C. Berkeley's "The Algebra of States and Events," *The Journal of Symbolic Logic, 20*, 286–287 (Sept., 1955).

in Burks and Wright [1], p. 1360; the difference lies in the fact that Church's "A's" and "B's" are independent of each other, whereas in the Burks-Wright definition of determinism each A_i has a certain relation to the corresponding B_i. Because of this relation between the two definitions, it follows directly that every transformation realized by a well-formed net is definable by a Church recursion. The converse may be shown by a net construction in which for each i a net is made for A_i and for B_i, and the outputs are combined to give A_i for $t = 0$ and B_i for $t > 0$.

It is perhaps worth noting that our decision procedure may be extended to give a method for deciding whether the transformations defined by a set of equations (Burks and Wright [1], p. 1358) are deterministic or not. This may be done by going through all the states $<I, S>$ and seeing if for each of these the equations yield a unique S'. If a given S is admissible (by the admissibility tree) and does not yield a unique S' for each I, then the net (i.e., the transformations it realizes) is not deterministic. There is, in any event, only a finite number of cases to consider, so the procedure is effective.

We remark finally that since monadic propositional functions of time may be used in describing net behavior, it might seem that known decision procedures for the monadic functional calculus directly apply here. However, the exact relation between quantifiers and net theory is not known, and in any event when quantifiers are used they required bounds, which are essentially dyadic (see Section 4.2).

2.3 *Representation by Nets, The Coded Normal Form*

We turn now to the representation of automata by diagrams (called automata nets) which show the internal structure of automata. For this purpose we need to correlate the binary digits zero and one used in the preceding subsection to the physical states of wires, junctions, or cells. On the *normal interpretation* zero and one are associated with the inactive and active states, respectively. A *dual interpretation* (zero to active, one to inactive) is also possible, and the two interpretations may be interrelated by the well-known principle of duality.

It is clear from the developments of the preceding subsection that we need net elements capable of performing two kinds of operations: truth functions and delays. For these purposes we adopt two distinct kinds of elements: switching elements for truth-function operations and a delay element for the delay operation.

Some standard logical connectives of the theory of truth functions are: $\cdot$, & (two representations of conjunction, "and"), $\vee$ (disjunction, "or"), $\downarrow$ ("neither-nor"), $|$ ("not both"), $\equiv$ ("if and only if"), $\supset$ ("if . . . then . . . "), and $-$, $\sim$, $'$ (three representations of negation, "not"). Circuits for realizing all of these are common. As is well-known, all truth functions may be constructed from the dagger ($\downarrow$) or from the stroke ($|$), so we shall in general assume sufficient primitive switching elements to realize these. Sometimes it is convenient to have an infinity of primitive switching elements, one for each truth function. Of course,

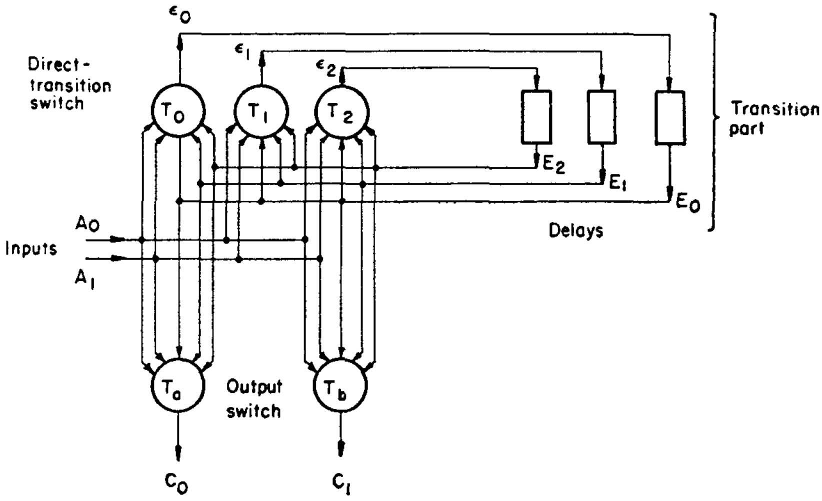

FIG. 1. Normal form net.

in practice complicated switching functions are realized by compounding simple switching elements, but by representing such circuits by single net elements we can separate the problem of compounding these circuits from other problems in net analysis (see, for example, fig. 1).

A *switching element* consists of a nucleus together with input wires and an output wire. The termini of these wires are called junctions. Switching elements may be interconnected in *switching nets* in ways to be discussed subsequently. For examples, see figure 1 and other figures of this paper. Propositional variables are associated with each junction of net. Corresponding to each switching element there will be an equation of the theory of truth functions which describes the behavior of that element. For example, if a conjunction ("and") element has the variables A_0 and A_1 attached to its input junctions (and wires) and the variable C_0 attached to its output junction (and wire), it realizes the equation $C_0(t) \equiv [A_0(t) \ \& \ A_1(t)]$, or, more succinctly, $C_0 \equiv (A_0 \ \& \ A_1)$. The theory of switching nets corresponds to the theory of truth functions and is well developed (see Shannon [12]; Burks and Wright [1]).

One aspect of the equation $C_0(t) \equiv [A_0(t) \ \& \ A_1(t)]$ needs discussion; it is that the value of the output is given at the same time t as the inputs. In the physical realization of a conjunction this, of course, cannot happen; the output will occur slightly later than the inputs. This suggests putting a delay in at the output of each switching element. Such a delay does in fact exist in each nerve cell. However, for purposes of theoretical analysis it is best to isolate the logical, nontemporal functions of automata from the temporal aspects of their behavior. Hence we can first construct the theory of switches, basing it on the theory of truth functions, and we can later augment this theory to deal with the additional complications brought in by delays. This organization of the subject has practical bearings as well as theoretical value, for to a certain extent the design of switches does and should proceed independently of the design of those parts of computers which produce the transitions from state to state. Hence our switching nets have no delays in them. When we come to formulate the rules governing their interconnection (formation rules for well-formed nets), we will take this factor of idealization into account and not permit interconnections that could lead to trouble because we have ignored it. Hence we will return to this topic at that time.

The *delay element* consists of a nucleus with an input and an output wire; see figure 1. It delays an input signal one unit of time; i.e., its input wire state at time t becomes its output wire state at time $t + 1$. We assume that its output wire is inactive (in the zero state) at time zero. If A_0 is the variable associated with its input and E_0 the variable associated with its output, its behavior is defined by the equations

$$E_0(0) \equiv 0$$

$$E_0(t+1) \equiv A_0(t).$$

Another way of expressing this is $E_0(t) \equiv (t > 0) \;\&\; A_0(t \dot{-} 1)$, where "$\dot{-}$" signifies the primitive recursive pseudosubtraction

$$x \dot{-} y = x - y \quad \text{if} \quad x \geqq y$$

$$x \dot{-} y = 0 \quad\quad \text{if} \quad x < y.$$

Each switching element corresponds to a symbol (or complex of symbols) of the theory of truth functions. We will introduce the symbol δ to correspond to the delay element. If the input and output to the delay element are A_0 and E_0 as before, then $E_0(t) \equiv \delta[A_0(t)]$ or, more succinctly, $E_0 \equiv \delta A_0$. Hence,

$$\delta[A_0(0)] \equiv 0$$

$$\delta[A_0(t+1)] \equiv A_0(t).$$

In itself the δ operator does not, strictly speaking, take us beyond the theory of truth functions (see Section 4.2), but the δ operator together with a cycle rule which allows an output of a net to be connected back to an input of the same net does take us beyond truth-function theory to quantifier theory (see Section 4).

We need now a set of formation rules such that all nets constructed by these rules represent automata, and all automata defined by characterizing tables and output tables may be represented by these nets. We will use the rules given in Burks and Wright [1], p. 1361, extending them to allow an arbitrary set of switching elements Σ.

DEFINITION 4: A combination of figures is a *well-formed net* (w.f.n.) relative to the set Σ if and only if it can be constructed by the following rules:

(1) A switching element or a delay element is w.f.

(2) Assume N_1 and N_2 are disjoint w.f.n. Then,

(a) the juxtaposition of N_1 and N_2 is w.f.;

(b) the result of joining functions $F_{q_1}, \cdots, F_{q_j}$ of N_1 to distinct input junctions $G_{p_1}, \cdots, G_{p_j}$ of N_2 is w.f.;

(c) the result of joining input junctions F_p and F_q of N_1 is w.f.;
(d) if all the wires connected to F_p of N_1 are delay-element input wires, then the result of joining any F_q of N_1 to F_p is w.f.

The ends of wires which do not impinge on a switching-element circle or a delay-element rectangle are called *junctions*. A junction with no output wires attached to it is called an *input junction*; all other junctions are called *internal junctions* (these are sometimes called *output junctions*).

One can label each junction of a net with a variable. We will usually use A_0, A_1, $\cdots$ for input junctions, C_0, C_1, $\cdots$ for *switch output junctions* (junctions driven by switching elements), and E_0, E_1, $\cdots$ for *delay output junctions* (junctions driven by delay elements). A well-formed net (diagram) with every junction labeled with a variable is called a *labeled net.* One can also label the input junctions with functional constants designating particular input functions (e.g., 000 $\cdots$, 111 $\cdots$, 0101 $\cdots$, 0100100001000001 $\cdots$), and the internal junctions with functional constants naming the functions they actually realize (e.g., if the inputs to a conjunction are labeled with 111 $\cdots$ and 010101 $\cdots$, the output should be labeled 010101 $\cdots$). The result is called a *net history*; cf. the concept of a net state in Burks, McNaughton, *et al.* [13], p. 207.

Consider the net of figure 1. Every net of this form (with arbitrary numbers of delays and switching elements) is well-formed (relative to a sufficiently rich set of switching elements) by our rules. We say that the net of figure 1 is in *normal form*. A normal form net is organized as follows. It has a *direct-transition switch*, fed by the net inputs and the delay outputs, and driving the delay inputs. It has an *output switch*, fed by the delay outputs and the net inputs, and not driving any delay elements.

Given a sufficiently rich set of switching elements, we can construct for each well-formed net a normal form net which behaves the same. We first place the delays of the original net in an array like that of figure 1. Then, for each delay element E_i, we analyze the original net to determine what switching element T_i will produce the same result at ϵ_i as is produced by the switching circuitry of the original net. In the same way we find those switching elements T_a, T_b, $\cdots$ whose outputs behave the same as the switch output junctions of the original net, or those switch output junctions we are particularly interested in. (The latter can be indicated by labeling them with triangles.) Similarly, given a set of switching elements Σ rich enough to represent all truth functions, we can translate a normal form net into a w.f.n. made of those switching elements; e.g., if Σ contains only the stroke element, we replace each T of figure 1 by an equivalent stroke-element switch. (Note that while a switching element T_i receives inputs from all the net input junctions and all the delay output junctions, its output need not depend on all of these. For example, if in the original net the input to delay element E_2 was the net input junction A_0, then T_2 has the property that $\epsilon_2(t) \equiv A_0(t)$ for all values of $A_1(t)$, $E_0(t)$, $E_1(t)$, and $E_2(t)$.) (Note also that any well-formed net can be arranged somewhat in the form of figure 1, if we allow the switches to be of other forms and allow the direct-transition switch to have junctions which do not drive delay inputs.)

At this point we wish to make two comments about our representation of switches. The first concerns a topic we have mentioned earlier, the fact that physically it takes time for information to go from the inputs of a switch to the output, while in calculating the behavior of a switch we assume that the output occurs at the same time as the input. The reason for this assumption is that in many applications the switching time is much less than the delay time, so the logic of switching is treated separately from the logic of delay. We wish our theory to accommodate this case. The reader can imagine a small delay in the output of each switching element, with extra delays put in at various places to make the phasing correct. He can then imagine that each unit delay of figure 1 is reduced by the accumulated amount of delay in the switch driving it, so the total delay from delay output back to delay output is one unit. (The concept of rank as defined in Burks and Wright [1], p. 1361, is useful here.) The concept of well-formed net has been so defined as to make this always possible, as is evident from figure 1. This way of regarding the matter conforms with practice in designing some machines; see De Turk *et al.* [14]. Those automata with delay built into each switching element (e.g., neural nets) can also be accommodated within our theory; they correspond to special cases of w.f.n. and can be defined by modifying the formation rules.

The second comment is connected with the fact that our switching elements represent the flow of information in only one direction; i.e., inputs and outputs are not interchangeable. There are many devices that permit information to flow in only one direction (vacuum tubes, transistors, etc.), but not all do; relays are one notable exception. Relay contacts permit information to flow in either direction, and hence bridge circuits can be made from them. Relays are electromechanical devices and hence are relatively slow. For this reason they are becoming less important as much faster electronic and solid-state devices become available and competitive in price. Further, because of the combination of a coil and contacts, relay automata present special problems, and no formation rules for them which take full account of all the uses that can be made of contacts and coils have been published. However, a new and promising device, the cryotron, also permits the information (in this case, current) to flow in either direction and hence can be used in bridge circuits (see Buck [15]). We will not attempt here to devise formation rules for all uses of relays, cryotrons, and whatever other devices there may be which are not unidirectional.

It should be pointed out, however, that every well-formed switching net can be realized by a relay and by a cryotron circuit. Since every truth transformation (i.e., every switching function) can be represented by a well-formed switch and vice versa (Theorem XII of Burks and Wright [1]), our diagrams do represent ways of realizing all truth functions with nonunidirectional elements. Since our diagrams represent a unidirectional flow of information, it follows that the power of relays and cryotrons to pass information in two directions does not add to their power to do logic. It does make a difference in the number of elements needed. Thus a relay bridge circuit may do a certain job more economically than a relay contact network in the form of a well-formed switch.

We return now to the problem of correlating w.f.n. and automata. As a first step we will define a set of state numbers $D_0, D_1, \cdots, D_q$. Each D will express the states of the delay output junctions. Let these junctions be labeled $E_0, E_1, \cdots, E_q$. Then D is the binary number $E_0{}^\frown E_1{}^\frown \cdots {}^\frown E_q$. Since a delay output is assumed to be zero at time zero, $D(0) = 0$. Let D_0 be this initial state, i.e., $D(0) = D_0$. We wish to justify this decision, but before doing so we need to discuss a question concerning the identity of an automaton.

We may run a machine from Monday to Friday, turn it off at 5:00 P.M. Friday and then turn it on again at 8:00 A.M. the following Monday. Should we regard it as one machine or two? There is a similarity between this and a human (automaton?) going to sleep at night; however, when a human wakes up in the morning he still remembers quite a bit of his past history, while often (though not always) a computer starts a new life every time it is turned on anew. To preserve the identity of the machine before and after the gap of inaction, we can think of some simple *ad hoc* device such as a special input cell whose sole function is to turn the machine on and off in such a manner that, when a machine is in operation, stimulating this input cell will put the machine into a unique initial state; such an operation is often called an initial clear. In this way we can preserve the identity of a machine through all the different runs it makes.

On this assumption there is only one initial state of an automaton, and we can identify it with the all-off or all-quiet state. Such an identification is natural since neurons, vacuum tubes, etc., are usually inactive when first turned on, and even if they are not this identification can be made by a suitable convention without much loss of generality. The situation is somewhat different if we choose to regard each machine run as a new automaton, but even here there will probably be a single initial state for all runs and it is convenient to identify it with the all-quiet state. Note that this does not commit us to identifying S_0 with D_0. In fact, we shall not always do so (the complete decoded net of Sec. 3.2 is an example). Hence one can handle other initial states by identifying D_0 with some value of S other than S_0.

We now proceed to establish the equivalence of w.f.n. and automata. We show first how to derive a characterizing table and an output table for each w.f.n. We translate the given net into a normal form net. Label the inputs of this normal form net $A_0, A_1, \cdots$ and let $I = A_0{}^\frown A_1{}^\frown \cdots$; for a two input net we would have, for example, $I_0 \equiv \bar{A}_0 \;\&\; \bar{A}_1$, $I_1 \equiv \bar{A}_0 \;\&\; A_1$, $I_2 \equiv A_0 \;\&\; \bar{A}_1$, and $I_3 \equiv A_0 \;\&\; A_1$. Label the delay inputs $\epsilon_0, \epsilon_1, \cdots$ and define $\Delta = \epsilon_0{}^\frown \epsilon_1{}^\frown \cdots$. Label the delay outputs $E_0, E_1, \cdots$; then $D = E_0{}^\frown E_1{}^\frown \cdots$. Let $T_0, T_1, \cdots$ be the truth functions realized by the direct-transition switch. Then we have

$$\epsilon_i(t) \equiv T_i[A_0(t), A_1(t), \cdots; E_0(t), E_1(t), \cdots]$$
$$E_i(t) \equiv \delta\, \epsilon_i(t)$$

for each i. Finally, we let D_0 be S_0 and each other D be an S, and thereby get a complete table. By the use of the admissibility tree we can construct the characterizing table. This procedure takes care of the transition part of a normal form net. To complete the analysis, we perform a similar construction for the switching elements of the original net, or for those switch outputs we are interested in as final outputs. Let T_a, T_b, $\cdots$ be the truth functions realized by these outputs. Then we have

$$C_j(t) \equiv T_j[A_0(t), A_1(t), \cdots ; E_0(t), E_1(t), \cdots]$$

for each j, and this gives us the output table.

A *coded normal form net* is a normal form net whose characterizing table is in coded normal form.

When the complete table is derived from a net, there will be a bit position for each input junction and each delay output junction. In this case the numbers m and n (of Section 2.1) are the numbers of input and delay output junctions, respectively. An $m - n$ automaton has then 2^{m+n} possible complete states and 2^n possible internal states. If all input states are considered, we then have $(2^n)^{(2^{m+n})}$ different $m - n$ automata complete tables. Many of these are the same except for the permutation of columns (i.e., of input and internal cells or junctions). Clearly, there is little significance whether a particular junction is labeled the 1st, 2nd, or the mth. In other words, if we can find a way of identifying one-to-one input junctions of two $m - n$ automata so that they behave the same, they are equivalent even though they may have different characterizing tables. Analogously, permutations among the labels for the delay output junctions make no essential difference. It follows that there are actually only $(2^n)^{(2^{m+n})}/(m!)(n!)$ rather than $(2^n)^{(2^{m+n})}$ distinct abstract $m - n$ automata complete tables. Similarly, characterizing tables which are obtainable from one another by permuting columns are to be identified. There will be fewer than $(2^n)^{(2^{m+n})}/(m!)(n!)$ distinct $m - n$ automata characterizing tables, for some of the distinct complete tables will differ only with regard to inadmissible states.

In designing the transition part of an automaton it is in general desirable to maximize the number of admissible internal states (i.e., to minimize the number of inadmissible states), since the total number of states is a rough measure of the parts needed for construction, while the capacity for doing different things is in general proportional to the number of admissible states. We will call an automaton *complete* if all states S are admissible. (A stronger condition would be that all states S are admissible relative to every initial state, instead of just the distinguished initial state S_0; we will not give to automata satisfying the stronger condition any special name.) If the number of admissible states of an automaton is less than 2^{n-1}, we can always replace the automaton by a simpler automaton by using the coded normal form. In a coded normal form characterizing table n is the least integer as large as or larger than the logarithm of N to the base two (similarly for m).

For automata with the same number of admissible states, it seems desirable to maximize the "recoverable" ones. Following Moore [16], p. 140, we shall call an automaton *strongly connected* if and only if it is possible to go from every admissible state S_i to every admissible state S_j (i may be equal to j). An alternative definition can be given in terms of an admissibility tree in which all occurrences of a given state S_i are identified. An automaton is strongly connected if and only if for any ordered pair of states $< S_i, S_j >$ (i may be equal to j), we can pass from S_i on the tree to S_j on the tree by a continuous forward route (plus backward jumps from a given state to the same state located lower on the tree). Since the possibility of repetition is important for an automaton, any admissible state which cannot be recovered adds rather little to the capacity of the automaton. Thus it would seem best in general to design a machine which is both complete and strongly connected.

Hence, from a practical point of view, complete, strongly connected, and coded normal form automata are the most important. For the theory of automata, however, many nets falling outside this class are of interest. In particular, we will find decoded normal form automata nets of interest in connection with the use of matrices to analyze nets.

It remains to show how to construct a well-formed net for any given characterizing table (or complete table) and output table. There are various ways of doing this, one of which is to identify S_0 with D_0 (if S_0 is not equal to zero, its value must be changed to zero) and to let every other S be a value of D. The general process of going from nets to tables is then just reversed. There are various ways of constructing the switches needed. Let us consider the matter with regard to the characterizing table $\ll I, S>, S'>$. A single column of S' is to be identified with a particular E_i (and ϵ_i). Delete all other columns of the S' part of the table. We then have a truth-table definition of our function T_i, such that

$$\epsilon_i(t) \equiv T_i[A_0(t), A_1(t), \cdots ; E_0(t), E_1(t), \cdots].$$

Given sufficient primitives, this can be realized by one switching element, as in figure 1. Given switches for "and," "or," and "not," it can be realized by using disjunctive normal form. Consider each row of the truth table. If ϵ_i is zero, do nothing; if ϵ_i is one, construct an element to sense $I \frown S$ of that row. The desired switch for ϵ_i is obtained by disjoining (using "or" on) all the outputs so obtained.

We have thus established our first theorem.

THEOREM 1: *Given a well-formed net, we can construct a complete table, a characterizing table, and an output table describing its behavior. Given a complete table, characterizing table, and an output table, we can construct a well-formed net realizing these tables.*

This theorem establishes the equivalence of automata and nets, and since nets are idealized representations of digital computers, it follows that for most theoretical considerations automata without any special sensing and acting organs can be viewed as digital computers.

We conclude this section by noting the similarity of well-formed net diagrams and flow diagrams used in programming. This similarity is what one would expect, since a net diagram describes the structure of a computer, and a flow diagram describes its behavior during a certain computation, and both symbolize recursive functions. While a program is stored in a computer, part of it (the coded representation of the operations) usually remains invariant through the computation; this means that during the computation not all states of the computer are used. For each such fixed program one could devise a special-purpose machine which would perform the same computation. This is a special case of the general principle that there is a great deal of flexibility with regard to what a machine is constructed to do versus what it is instructed to do. This suggests that there should be one unified theory of which the theory of automata structure and the theory of automata behavior (i.e., the theory of programming) are parts.

Each program is in effect a definition of a recursive function. Since there is no effective way of deciding whether two definitions define the same recursive function, there is no effective way of deciding whether two programs will produce the same answer. Two different programs, each finite, may nevertheless produce the same answer because the feedback from the computation may be different in the two cases.

REFERENCES

1. Burks, Arthur W., and Jesse B. Wright, Theory of Logical Nets, *Proc. IRE, 41:* 1357-1365 (1953).
2. Burks, Arthur W., and Irving M. Copi, The Logical Design of an Idealized General-Purpose Computer, *J. Franklin Inst., 261:* 299-314 and 421-436 (1956).
3. Shannon, Claude, Computers and Automata, *Proc. IRE, 41:* 1234-1241 (1953).
4. Rochester, N., J. H. Holland, L. H. Haibt, and W. L. Duda, Tests on a Cell Assembly Theory of the Action of the Brain, Using a Large Digital Computer, *IRE Trans. on Information Theory*, 1956, pp. 80-93.
5. Turing, A. M., On Computable Numbers, with an Application to the Entscheidungsproblem, *Proc. London Math. Soc.* (Series 2), *42:* 230-265 (1936-37), with a correction, *ibid., 43:* 544-546 (1937).
6. Kleene, S. C., *Introduction to Metamathematics*. New York: D. Van Nostrand Company, Inc., 1952.
7. Wang, Hao, A Variant to Turing's Theory of Computing Machines, *J. Assn. Computing Machinery 4:* 63-92 (1957).
8. Wang, Hao, Universal Turing Machines: An Exercise in Coding (to be published).
9. von Neumann, John, The General and Logical Theory of Automata, pp. 1-41 in *Cerebral Mechanisms in Behavior*, John Wiley and Sons, 1951.
10. Kemeny, John G., Man Viewed as a Machine, *Scientific American, 192:* 58-67 (1955).
11. Kleene, S. C., Representation of Events in Nerve Nets and Finite Automata, pp. 3-41 in *Automata Studies*, edited by C. E. Shannon and J. McCarthy, Princeton Univ. Press, 1956.
12. Channon Claude, A symbolic analysis of relay and switching circuits, *Trans. AIEE, 57:* 713-723 (1938).
13. Burks, Arthur W., Robert McNaughton, Carl H. Pollmar, Don W. Warren and Jesse B. Wright, Complete Decoding Nets: General Theory and Minimality, *J. Soc. Ind. Appl. Math., 2:* 201-243 (1954).
14. De Turk, J. E., A. L. Garner, J. Kautman, A. W. Bethel, and R. E. Hock. *Basic Circuitry of the MIDAC and MIDSAC*. Ann Arbor: Univ. of Mich. Press, 1954.
15. Buck, D. A., The Cryotron—A Superconductive Computer Component, *Proc. IRE, 44:* 482-493 (1956).
16. Moore, Edward F., Gedanken-Experiments on Sequential Machines, pp. 129-153 in *Automata Studies*, edited by C. E. Shannon and J. McCarthy, Princeton Univ. Press, 1956.
17. Hilbert, D., and P. Bernays. *Grundlagen der Mathematik*. Vol. I. Berlin: Springer, 1934.