Modular Design of Asynchronous Circuits Defined by Graphs

RENÉ DAVID

Abstract—**An universal cell (CUSA) permits one to realize any asynchronous automaton by associating such a cell with a state and a connection with a transition. There is no internal variable assignment and all kinds of hazards are avoided. In the first part of the paper we shall present the principle of the CUSA and the synthesis method using flow tables. It is an original method which enables one to reduce the number of cells and useful connections. In the second part, the method is adapted to a synthesis from a flow graph. In this graph, a node is associated with a state, not necessarily a total state, and a branch with a connection. This synthesis method is easy and systematic. CUSA are now being produced by industry and are available as IC's.**

Index Terms—**Asynchronous circuits, description by graphs, modular networks, synthesis from graphs, synthesis without assignment, universal cell CUSA.**

I. Introduction

THE SYNTHESIS of asynchronous automata is usually reduced to the well known synthesis of combinational networks. Huffman's method [1], which consists of describing the function by a flow table, of an assignment of the internal variables, then of a combinational synthesis of these internal variables and outputs, is an illustration of this synthesis. This method presents difficulties in assigning the internal variables while satisfying the adjacency conditions and eliminating all of the hazards.

Since then, interesting solutions for carrying out an assignment and for eliminating hazards have been suggested by numerous authors. However, it remains difficult to apply these solutions to problem of increasing complexity. Different cellular approaches to automata synthesis have been carried out [2]–[7]. In the majority of cases they greatly emphasize the combinational aspect, and they all have limitations at the level of either the structure or the complexity of the automata. Low and Maley's work [5] is certainly the closest to the present work. However, there are two types of cells, for total stable states and for unstable states, respectively, and no simplification methods are given. The method introduced here has been developed in order to obtain a systematic and simple synthesis without assignment. In order to do this a cell is associated with each state. The essential contribution consists of introducing a cell with a particular delay insuring that the networks obtained do not show any first-order hazards, and also of an original synthesis and simplification method.

Initially, a method of synthesis was carried out by using a flow table as a description of the automaton [9], [10]. That work will be summarized in Section II. But the use of flow tables is limited and it appeared necessary for large systems to develop the synthesis starting immediately from a graph describing an automaton more concisely [11]. In Section III, synthesis from a graph is presented; each state, represented by a node in the graph, is not necessarily a total state; some inputs may vary without any change of state.

This description and the synthesis which is deduced are particularly well adapted to industrial type automata which are rather complex.

II. Principle of Synthesis from Primitive Flow Tables [8]–[10]

In this section, the essential elements of synthesis will briefly be summarized. For more details, the reader is referred to the works cited. It is assumed here that only one input variable changes at a time.

A. Principle of the CUSA and Canonical Synthesis

The canonical synthesis principle with the help of CUSA (in French Cellule Universelle pour Séquences Asynchrones) consists of associating a cell with each total stable state and a wire between two cells with each unstable state or transition. The output of a cell is at level 1 if the system is in the corresponding state.

Let us consider Fig. 1(a), as part of a primitive flow table, and Fig. 1(b) the corresponding part of its primitive flow graph[1], associated with the state b. Near each branch of this graph the input variable whose variation from 0 to 1 produces the corresponding transition is marked. For example, passing from state d to state b corresponds to the variation of x_1 from 0 to 1, and passing from state b to state a corresponds to the variation of x_1 from 1 to 0, i.e., to the variation of x_1' from 0 to 1. Thus branches $d \rightarrow b$ and $b \rightarrow a$ are, respectively, labeled x_1 and x_1'.

We shall associate a cell B, the output of which is y_b, with the state b. Let us see what *logical* and *technological* conditions have to be satisfied by this cell B.

Logical Conditions: To reach the state b it is necessary to 1) be in one of the predecessor states a, d, or e; that is to say, $y_a + y_d + y_e = 1$, and 2) reach the column where b is stable; that is to say $x_1x_2' = 1$.

When the system has reached the state b, $y_a + y_d + y_e$

Manuscript received September 1, 1975; revised June 1, 1976.
The author is with the Laboratoire d'Automatique, I.N.P. Grenoble, BP 15,38040 Grenoble-Cedex, France.

[1] A primitive flow graph is a graph in which each node is associated with a total state, as a primitive flow table is a flow table in which each row is associated with a total state.

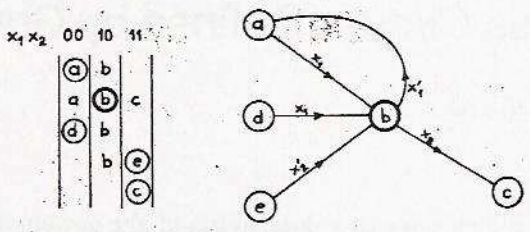

Fig. 1. Part of a primitive flow table and of the corresponding flow graph.

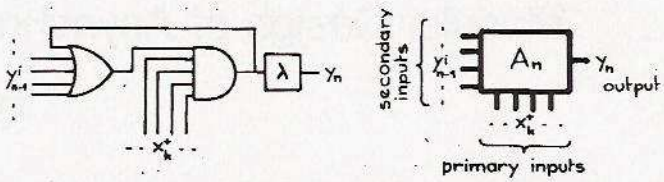

Fig. 2. CUSA. (a) Realization with OR, AND gates. (b) Symbolic representation.

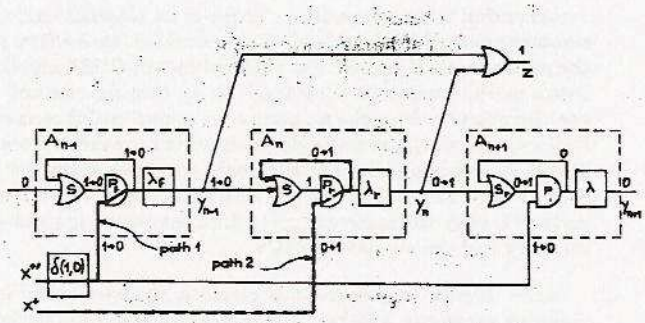

Fig. 3. Transition from state a_{n-1} to state a_n.

becomes 0 but y_b has to keep the value 1 until the input state changes, that is to say x_1x_2' becomes 0. So we get,

$$y_b = (y_a + y_d + y_e + y_b)x_1x_2'.$$

In the general case, let us consider the state a_n, represented by the cell A_n, the output of which is y_n. The output of the predecessor cell A_{n-1}^i is denoted as y_{n-1}^i and state a_n is stable for the input state $x_1^+ = \cdots = x_k^+ = \cdots = 1$ (x_k^+ corresponds to either x_k or x_k', and $x_k^{+'}$ to either x_k' or x_k, respectively).

We get the logical conditions

$$y_n = \left(\sum_i y_{n-1}^i + y_n\right) \prod_k x_k^+.$$

The CUSA may be realized as shown in Fig. 2(a). We shall now discuss the delay λ.

Technological Conditions: The network has exactly the same behavior at each transition: one CUSA takes the value 1, while the preceding one comes to the value 0. To avoid all hazards, it is sufficient to insure that each transition behaves correctly.

Let us consider the transition from state a_{n-1} to state a_n (Fig. 3: other inputs that x are constant and are not represented). The possible malfunctions are the following: the 1 level is not propagated to the CUSA A_n which keeps the 0 value; or it is propagated too quickly and a 1 value appears on y_{n+1}; a hazard appears on the output z.

These possible malfunctions are avoided by a delay λ introduced at the output of each cell. Let us see what must be the properties of λ. In Fig. 3 each CUSA is made up of two gates S and P and a delay λ. We shall denote the associated propagation times by subscripts r and f for a rise and a fall of their outputs, respectively. The delay between the rising input x^+ and its complement $x^{+'}$ is denoted by $\delta(1,0)$.

1) To be sure that y_n has enough time to achieve the value 1, it is necessary that path 1 be longer than path 2 (Fig. 3).

$$\delta(1,0) + P_f + \lambda_f > P_r$$

$$\lambda_f > P_r - \delta(1,0) - P_f = \lambda_{f0}.$$

2) To avoid the possibility of y_{n+1} taking a transient value 1, we find, similarly

$$\lambda_r > \delta(1,0) - P_r - S_r = \lambda_{r0}.$$

3) The continuity of an output z, the value of which is 1 in states a_{n-1} and a_n, is insured if

$$\lambda_f - \lambda_r > P_r - \delta(1,0) - P_f = \lambda_0.$$

The delay λ which is in turn λ_r and λ_f has to satisfy simultaneously the three preceding inequalities. The values of λ_{r0}, $\lambda_{f0} = \lambda_0$ depend on the conditions of use (possible delay between an input x and its complement x', fan-in, fan-out, ...) and of the technology (characteristics of the components, dispersion of these characteristics, ...). Considering the worst case, and a security margin, we realize a delay which merely delays with a different value for the rise (λ_r) and the fall (λ_f) of its input level.

When the CUSA are carried out technologically, including the delay designed once and for all, we have a canonical wiring which consists of interconnecting identical CUSA as shown in Fig. 2.

From a primitive flow table, the *canonical wiring* consists of associating a CUSA with each state and in wiring such that (see Figs. 4 and 5):

1) The primary inputs are associated with the column of the corresponding state.

2) The secondary inputs correspond to the outputs of the predecessor CUSA.

3) Each output is realized by an OR gate.

Remarks:

1) The CUSA may be realized with an OR and an AND gates as shown in Fig. 2, or with 2 NOR gates (one has to replace x_k^+ by $x_k^{+'}$), or with 2 NAND gates (then the active level is 0 and a network output is realized by a NAND gate).

2) CUSA are now available from the SESCOSEM French company as TTL integrated circuits [13]. The delays are integrated in DIL packages and the designer of sequentials networks need not worry about these delays. One package is available with 2 CUSA, another with 1 CUSA, with 3 and 3 or 6 and 7 primary and secondary inputs, respectively. These packages allow realization of most

x_1x_2	00	01	11	10	z_1z_2
	(a)	b	-	d	0 1
	a	(b)	c	-	1 1
	-	b	(c)	d	1 1
	a	-	e	(d)	0 0
	-	f	(e)	d	0 0
	a	(f)	e	-	0 0

Fig. 4. Flow table of machine M_1.

of the practical systems without fan-in problems. If necessary, extensions may be done with gates (taking care of added delays on primary inputs).

3) Within some constraints on the inputs, the delay λ is not useful. This is particularly interesting for medium- or large-scale integration.

If each input variable x and its complement x' is available from the output of a flip-flop made with 2 NAND gates, the delay is not useful. As shown in Fig. 6, the falling level (in turn x and x') is always delayed by 1 unit (1 unit = delay of 1 gate) from the rising level (in turn x' and x) i.e., $\delta(1,0) = 1$ unit for *every* transition. With $\delta(1,0) = P_r = P_f = S_r = 1$ unit, the technological conditions give $\lambda_{f0} = \lambda_{r0} = \lambda_0 = -1$ unit, so $\lambda_r = \lambda_f = 0$ is a solution. The same result may be obtained with CUSA carried out with 2 NOR gates or 2 NAND gates if the input variables are available from flip-flops made with 2 NOR gates or 2 NAND gates, respectively.

4) This has been used for an MSI manufacture in MOS technology [12]. The chip contains 15 inputs (each of them associated with a flip-flop made with NOR gates), 30 CUSA (made with 2 NOR gates, without delay), and 10 outputs. Between the input circuits at the top of the chip and the output gates at the bottom, the CUSA covers a *vertical* strip. Three masks are independent of the network and the fourth specifies the connections corresponding to the desired network. These connections are *horizontal* metallizations on three fieds corresponding to primary inputs, secondary inputs, and outputs realization, respectively.

B. Simplification Method

There are 2 types of simplification. The first type consists of eliminating certain connections and eventually some CUSA utilizing source states. The second type of simplification consists of gathering some states to be represented, under certain conditions, by only one CUSA.

Suppression of Connections and Cells: Let us consider the machine M_1 (Fig. 4) and assume that the input state is $x_1 = 1, x_2 = 0$, i.e., $x_1x_2' = 1$. The system is certainly in state d, because this state is alone in its column, no matter what the sequence must have been before. So it is possible to replace all the secondary inputs of the CUSA D by a permanent level 1. Let us call d a "source state" because y_d takes the value 1 as soon as the corresponding input state is reached, independently of what happened before. State a is also a source state. Suppressing all secondary inputs of CUSA A and D in Fig. 5 leads to Fig. 7.

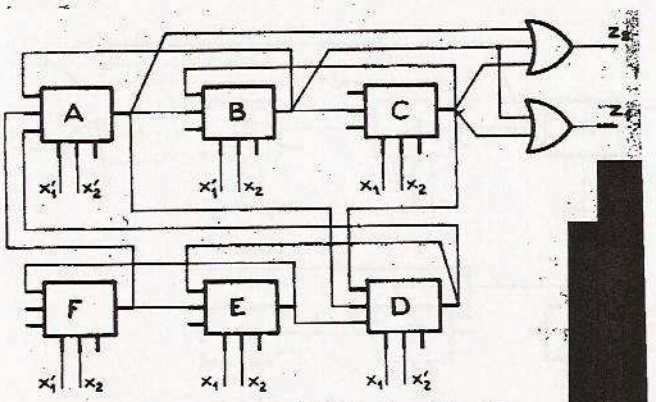

Fig. 5. Canonical wiring of machine M_1.

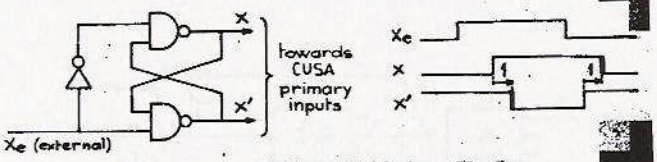

Fig. 6. Input variable available from flip-flop.

We see, now, in Fig. 7, that the network is disconnected and that cells D, E, F are unuseful for realizing outputs z_1 and z_2. These CUSA may therefore be cancelled.

Remark: States which are alone in their column are source states of the first kind. Other source states have the same property that every secondary inputs may be replaced by a permanent 1 level. This property, based upon the new notion of *hypovalence,* is not discussed here because it is not used for synthesis from graphs. The reader is referred to [9] or [10] for further information.

State Gathering: Let us see under what conditions two or more states can be represented by only one CUSA. Assume the machine M_1 is in the state b (see Fig. 7) and the outputs are $z_1 = z_2 = 1$. If x_1 takes the value 1, we pass from state b to state c and the outputs are always $z_1 = z_2 = 1$. Input x_1 may come back to 0, b is reached again, and so on. So changing x_1 keeps the system in states $\{b,c\}$ without changing the outputs and we can represent the double state bc by only one CUSA BC on which neither x_1 nor x_1' are wired. The new state bc is insensitive to the input variable x_1. We get the simplified network of Fig. 8.

The following has been shown in [9]:

Gathering conditions: One can get an equivalent cellular realization by representing two states a and b by only one CUSA if the following four conditions are fulfilled:

1) They produce compatible outputs.

2) Their successor states are compatible.

3) They are only differentiated by one input x_i.

4) Let $x_i^\dagger = 1$ in state a. No transition from any other state except that of state b towards state a can be achieved by $x_i^\dagger$ (and vice-versa).

Two double states can be gathered into a quadruple state if they satisfy the gathering conditions, and so on.

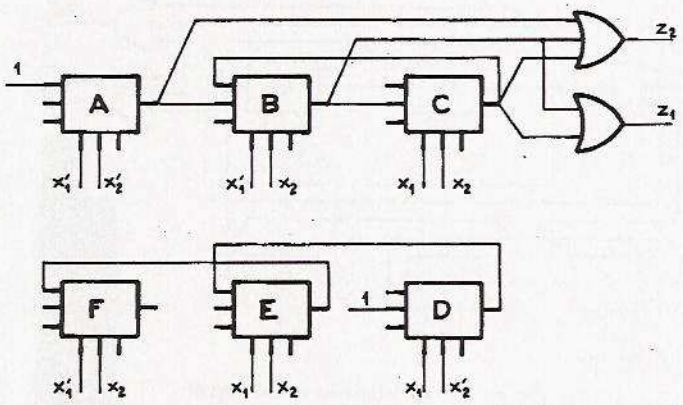

Fig. 7. Suppression of secondary inputs of A and D (source states). Machine M_1.

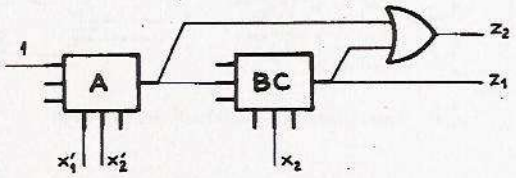

Fig. 8. Simplified network for machine M_1.

Remark: The gathering conditions 1) and 2) correspond to the classical state merging conditions to keep a Moore machine form. The conditions 3) and 4) are needed for CUSA networks.

III. Synthesis from Graphs

This synthesis is easily applied to automata that can also be described by flow tables, but it is especially well adapted to industrial automata. The term "industrial automata" perhaps does not constitute an accurate class, but within this context it applies to systems which have many inputs, many outputs, and which are very incompletely specified (many input combinations being physically impossible). Starting from specifications, such a system is easier to define by a graph than by a large and almost empty flow table. The nodes of this graph usually correspond to multiple states already gathered. The most important simplifications for these systems result from state gathering.

A. Example of Synthesis from a Primitive Flow Graph

The analysis of the following example will help us to understand the more general case of synthesis from a nonprimitive graph. Let us consider again the machine M_1 of Fig. 4. It can be represented by the primitive flow graph of Fig. 9(a) with the notation given in Section II-A. Next to each state, the outputs which have the value 1 in that state are marked.

From this primitive flow graph of Fig. 9(a) we can get the canonical wiring by applying the following rules (which will be later generalized to a nonprimitive graph).

Rules of Wiring from a Graph: A CUSA is associated with each node a. They are wired: 1) as primary inputs; the incoming variables and the complement of the outgoing variables concerning state a; 2) as secondary inputs; the outputs of the predecessor CUSA. Each output is realized by an OR gate.

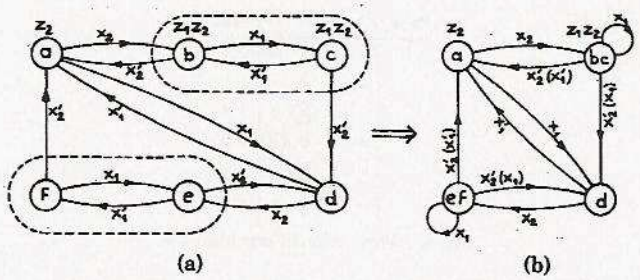

Fig. 9. (a) Primitive flow graph of machine M_1. (b) Gathering of states.

As an example let us consider the primitive flow graph of Fig. 9(a) and the corresponding network of Fig. 5. The cell F is associated with the state f. The incoming variable of f (near the branche entering f) is x_1'. The outgoing variables of f are x_1 and x_2'. As a result, the primary inputs of F are x_1' and x_2. Branch $e \rightarrow f$ is associated with the connection output of $E \rightarrow$ the secondary input of F.

Some simplifications are possible on the primitive graph. Let us first consider the gathering of states b and c of Fig. 9(a). They can be replaced by the state bc of Fig. 9(b) since they produce the same output and the input x_1, to which bc is insensitive, is neither an incoming nor an outgoing variable of state bc. Indeed, incoming and outgoing variables are wired on a CUSA (not complemented or complemented), while an insensitive variable is not wired. The input variable x_1 which can change from 0 to 1 or from 1 to 0 in state bc is noted near a loop. We have now two transitions from bc by the outgoing variable x_2'. These two transitions towards a and d cannot be simultaneous and depend on the value of x_1 which must be 0 and 1, respectively. They are denoted (x_1') and (x_1), respectively, for the two transitions considered. The variable (x_1') conditioning the transition $bc \rightarrow a$ is not considered to be an outgoing variable of bc because it is not necessary to have $x_1' = 1$ to leave bc, but it is considered as an incoming variable of a, because it is necessary to have $x_1' = 1$ to reach the state a (a generalization is given in Section III-B). States e and f can also be gathered. From the graph arrived at in Fig. 9(b) we can get a network of 4 CUSA by applying the preceding rules of wiring.

Let us now see if some states of Fig. 9(b) are source states. To be stable, the incoming variables of state a must be 1 and its outgoing variables must be 0, i.e., $x_1'x_2' = 1$. Let us call $M_a = x_1'x_2'$ the characteristic monomial of a. We then get, in the same way, $M_{bc} = x_2$, $M_d = x_1x_2'$, and $M_{ef} = x_2$. Let us assume that $M_a = x_1'x_2' = 1$, then $M_{bc} = M_d = M_{ef} = 0$. So if $x_1'x_2' = 1$ the machine is certainly in state a. It is a source state. We find that d is also a source state. Using this property, we get the graph of Fig. 10 with the following conventions: a source state is surrounded by a square; each branch reaching a source state is deleted; to

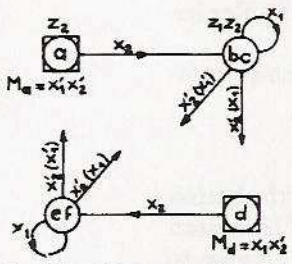

Fig. 10. Simplified graph of machine M_1.

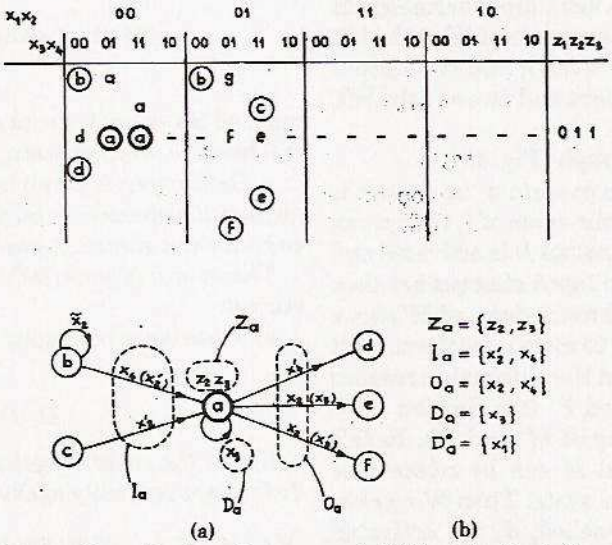

Fig. 11. Example of notation on a graph. (a) Part of a flow table: one state by row. (b) Corresponding part of the flow graph: one state by node.

determine which are the primary inputs to be wired, we note the characteristic monomials of the source states.

The part $\{d,ef\}$ of the graph is no longer useful because the corresponding CUSA are not used to produce any output, and they do not lead to any useful cells. Applying the wiring rules to the subgraph $\{a,bc\}$ of Fig. 10, we get the simplified network of Fig. 8.

B. Description of a Nonprimitive Graph

The proposed graph is thus formed as follows: near each node one notes for the corresponding state, the 1 output variables and the undetermined input variables; near each branch the input variable whose rise makes the corresponding transition is marked, and, eventually, input variables whose values enable the transition. These considerations are illustrated in Fig. 11 in relation to state a. When the system is in state b, the passage from 0 to 1 of x_4 (the variable which provokes the transition) leads to state a if $x_2' = 1$ (this variable which conditions the transition can be monomial). In state a, x_3 can change without the change of state.

Notation: Let us define some set of variables (see Fig. 11, for example):

Z_a is a set of unitary *outputs* in state a. $Z_a = \{z_2,z_3\}$.

I_a is a set of *incoming input variables* of state a. $I_a = \{x_2',x_4\}$ is the union of variables initiating the arrival to state $a\{x_2',x_4\}$, and of the variables conditioning the arrival to state $a\{x_2'\}$.

O_a is a set of *outgoing input variables* of state a. $O_a = \{x_2,x_4'\}$. These are the variables initiating departure from state a.

D_a is a set of *undetermined input variables* in state a. $D_a = \{x_3\}$. These are the variables which may be sometimes 0 and sometimes 1 in the same state a.

D_a^0 is a set of *invariant input variables* in state a. $D_a^0 = \{x_1'\}$.

One can also define some sets with complemented variables noted, for example, $I_a' = \{x_2,x_4'\}$, and other sets insensitive to the sign of the variables, for example, $I_a^* = \{x_2,x_4\}$.

If a variable x_i of O_a appears on several branches exiting from a, we call $O_a^{x_i}$ a set of monomials conditioning the transitions by a variation of x_i. In the example of Fig. 11:

$$O_a^{x_2} = \{x_3,x_3'\}$$

D_a^{0*} is a set of variables which do not belong to $I_a^* \cup O_a^* \cup D_a$. For example (see Fig. 11), x_1 has the value 0 during every transitions towards a, in state a, during every tran-

sitions from a. State a is insensitive to the variables belonging to $D_a \cup D_a^{0*}$.

Let us now examine an example of establishing a nonprimitive graph from specifications.

Example:

1) Specifications (see Fig. 12).

Let the wagon W be on contact P. As soon as the button M is pressed, the wagon goes to the right. If M is pressed long enough for the wagon to leave contact P, it goes to point Q and immediately comes back to P. When again reaching P, W stops, if M is not pressed, or starts a new cycle if M is pressed. This system has 3 input variables: M (starting pushbutton), P and Q (presence of W at the left and at the right contacts, respectively); and two output variables: R and L (move to the right and move to the left, respectively).

2) Establishing of the initial graph (Fig. 13).

At the start, let the system be in state a: no output is activated. As soon as M takes on the value of 1, the system passes into state b; in this state, output R is activated and the wagon goes to the right. Two input changes are then possible: either M is released, the branch denoted M' shows that the system then comes back to state a, or the contact P is left before releasing of M and the automaton reaches state c through the branch noted P' (see Section III-F about the simultaneous changing of M' and P'). In this state, R is always activated and M can be released or pushed again without a change of state. Then W reaches the contact Q. For the state reached, d, the activated output is now L; M is always undetermined and, going towards the left, the wagon will leave contact Q. Q can take on the value 0 without a change of state and is indicated as an undetermined variable for this state, d. When W reaches the contact P there are two possibilities: either $M = 0$ or $M = 1$. The two corresponding transitions towards a and b are marked $P(M')$ and $P(M)$, respectively.

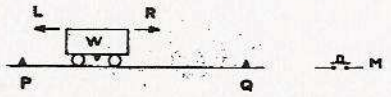

Fig. 12. Control of a wagon (machine M_2).

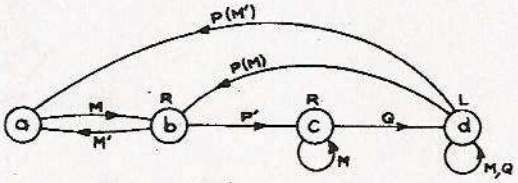

Fig. 13. Initial graph of machine M_2.

C. Functional Graph: Wiring

This is the most important part of the design. The initial graph is not unique, since a nonprimitive flow table is not unique, and it can contain some states gathered regardless of the gathering conditions of Section II-B. We shall now examine the correct construction methods working from a graph.

For a good understanding of the properties below, let us comment on Fig. 14.

Fig. 14(a): This corresponds to an ordinary transition, as every transition in a primitive flow graph. The input variable x_1, the outgoing variable of state a is wired as x_1' on A, the incoming variable of state b is wired as x_1 on B.

Fig. 14(b): The input variable x_2, undetermined in states a and b, is not wired on the corresponding cells.

Fig. 14(c): x_2 is undetermined in state a, thus not wired on A. The transition $a \rightarrow b$ is conditioned by x_2. This means that the changing of x_1 leads to state b, but only if $x_2 = 1$. Thus x_2 is wired on the CUSA B, and B can only be reached if $x_1x_2 = 1$. From state a, a change of x_1 when $x_2 = 0$ leads to another state, c in our example.

Definition: A graph is said to be *deterministic* if for any input sequence, from any initial state, the automaton *cannot exist simultaneously in two states.*

Theorem 1: A graph is deterministic if and only if for any state a:

a) A variable belonging to D_a does not appear in O_a^*, i.e.,

$$D_a \cap O_a^* = \phi.$$

b) For the same outgoing variable, the monomial conditions are mutually exclusive, i.e.,

$$\forall x_i^+ \in O_a, \qquad \forall m_j, m_k \in O_a^{x_i^+}, \qquad j \neq k: \quad m_j \times m_k \equiv 0.$$

The proof is quite obvious because condition a) expresses that the automaton cannot exist simultaneously in state a and in one of its successor states, and condition b) expresses the fact that the automaton cannot go simultaneously into two successors of state a.

Definition: A graph is said to be *functional* if, by applying the *rules of wiring* (Section III-A) to that graph, one can obtain an automaton which is exactly in *one state,* for any specified input sequence from any initial state.

Theorem 2: A graph is functional if:

a) It is deterministic (the two conditions of theorem 1 will be referred as (1-a) and (1-b)).

b) For any state a, a variable belonging to D_a does not appear in I_a^*, i.e.,

$$D_a \cap I_a^* = \phi.$$

One can prove this theorem by showing that the gathering conditions (Section II-B) are satisfied for the states gathered in state a, if this state a satisfies the conditions a) and b). Condition (1) is automatically fulfilled because only one output state can be associated with a node of the graph. Condition (2) expresses determinism; condition (a) insures this determinism. Condition (3), when applied to successive gathering, implies that *a variable belonging to I_a^* always appears with the same sign and a variable belonging to O_a^* always appears with the same sign but opposite to the first one,* i.e.,

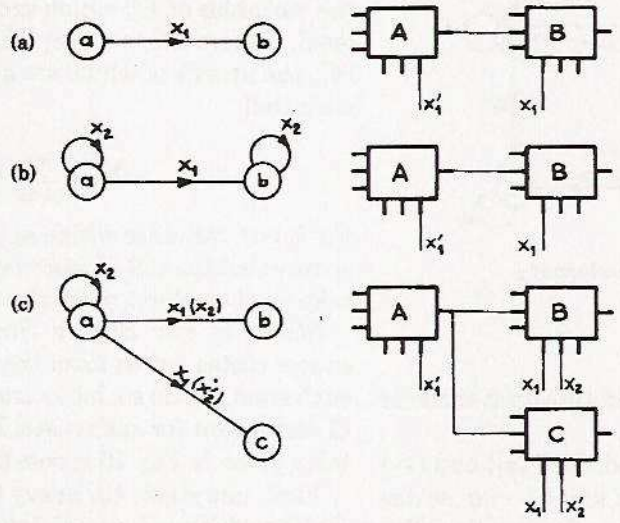

Fig. 14. Parts of graphs and associated wirings.

c) $x_i^{\dagger} \in I_a \cup O'_a \Rightarrow x_i^{\dagger'} \notin I_a \cup O'_a$.

If condition c)—very useful for the designer to avoid faults—is not satisfied for state a, some input variable must have the values 0 and 1, so it is an undetermined variable, and condition b), or condition (1-a) is not satisfied. Condition (4) is fulfilled if condition b) is satisfied.

Let us remark that condition a) about determinism is concerned with a correct description, independently of the synthesis, while condition b) is needed by the proposed synthesis method.

Obtaining a Functional Graph from an Initial Graph: If the *graph is not deterministic*, it may be necessary to return to the original specifications in order to describe an initial deterministic graph (in effect, one cannot guess at the real functioning).

If the *graph is deterministic* (this is the usual case if the designer has not made any mistakes), condition b) may not be satisfied. It is then necessary to "divide" (see below) into two the states which do not comply with this condition. Returning to the specifications is not useful here, but does allow one, in some cases, to avoid nonexistant transitions. Let us see, in an example, how to divide a state.

Application to the machine M_2: One can check that the graph of Fig. 13 is deterministic but not functional: indeed $Q \in I_d^* \cap D_d$. This state, d, is divided into two states d_1 and d_2, as shown in Fig. 15.

The passage from one to the other of these two new states is accomplished by variation of Q' (it is clear from specifications that a branch $d_2 \rightarrow d_1$ denoted as Q, cannot exist). Associated with states d_1 and d_2 are the same outputs variables as with d (i.e., $\{L\}$) and the same undetermined variables exept Q (i.e., $\{M\}$). A transition from c to d would be replaced by two transitions towards d_1 and d_2 in the general case, but here it can be a transition towards d_1, in order to satisfy condition c). In the general case each transition from d would be replaced by two transitions from d_1 and d_2 if they are compatible with the functionality conditions. But in our example, it is clear from the specifications that the transitions noted as $P(M')$ and $P(M)$ can only be achieved from d_2.

The graph of Fig. 15 is functional and one can directly make the wiring from this graph, but one can also obtain a simpler network.

Let us remark that a number of sufficient divisions always leads to a functional graph. Indeed, the limit is the primitive graph which is always functional.

D. Gathering of States: Irreducible Graph

When a functional graph is obtained, one can eventually reduce it by gathering 2 or more states. The only condition is that the new graph be functional. Let a group of states be neighbors if for any pair of states of this group there exists at least one chain leading from one to the other and passing only through the states of this group.

In practice, the *gathering conditions* for 2 or more neighbor states on a graph are as follows:

a) they produce the same output,
b) none of the incoming or outgoing variables of the new multiple state may change inside this state,
c) the outgoing conditions, which may eventually be added, leave a graph still functional.

The graph of Fig. 15 is said to be an *irreducible* graph because no further gathering is possible.

Let us study as an example the gathering of states b and c of machine M_1 (refer again to Fig. 9). The last 3 conditions are satisfied:

condition a): $Z_b = Z_c = \{z_1, z_2\} = Z_{bc}$
condition b): $I^*_{bc} \cap D_{bc} = \{x_2\} \cap \{x_1\} = \phi$
$O^*_{bc} \cap D_{bc} = \{x_2\} \cap \{x_1\} = \phi$
D_{bc} includes the undetermined variables

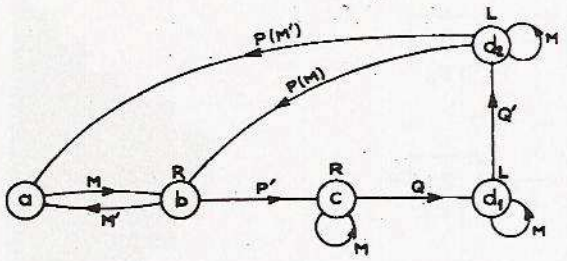

Fig. 15. Functional graph for machine M_2.

for b, those for c, and the input variable leading from b to c.

condition c): after addition of conditions (x_1') and (x_1) on branches $bc \rightarrow a$ and $bc \rightarrow d$, states a and d always satisfy the functionality conditions.

In general, one can also gather some states which are not neighbor states. One must then account for an invariant variable; if it has opposite signs in two states a and b, it is then undetermined for the new state, ab.

E. Source States: Simplified Graph

This is a generalization of what has been presented in the example of Section II-A.

Definition: A *characteristic monomial* of a state u is the monomial which characterizes the set of input states for which state u is stable. Let us denote it by M_u.

Consequently, M_u consists of all the input variables which do not belong to D_u, i.e., the incoming variables with their sign, the outgoing variables with the opposite sign, and the invariant variables with their sign. Equivalently,

$$M_u = \prod_{x_i^+ \in I_u \cup O_u' \cup D_u^0} x_i^+.$$

For example, one has for the state a of Fig. 11:

$$I_a = \{x_2', x_4\}, \qquad O_a' = \{x_2', x_4\}, \quad \text{and} \quad D_a^0 = \{x_1'\}.$$

Thus $M_a = x_1' \cdot x_2' \cdot x_4$ is a monomial which characterizes the 2 columns where state a is stable.

Property: State u is a source state of the first kind if and only if its characteristic monomial has a null product with the characteristic monomials of each of the other states of the automaton. This condition expresses the fact that state u is the only stable state for the set of input states, characterized by M_u. If the set of all machine states except u is denoted by Ω_u, one can write:

$$u \text{ is a source state of the first kind} \Leftrightarrow M_u\left(\sum_{v \in \Omega_u} M_v\right) = 0.$$

Simplified Characteristic Monomial: For a source state u, the wiring of all the variables of M_u as primary inputs ensures that the cell U will only assume the value 1 when the input state will correspond to state u. M_u contains the variables of I_u and O_u' which are indispensable, as well as the variables of D_u^0 which are not necessarily useful. Indeed, if there exists a simplified characteristic monomial M_u^*, the literals of which are a subset of the literals of M_u, such that

$$M_u^*\left(\sum_{v \in \Omega_u} M_v\right) = 0$$

the input variables which appear in M_u^* are sufficient to ensure that the cell U, associated with the state u, will not take on the value 1 when the system is in another state.

Study of the Source States: In order to find the source states, let us form the characteristic monomial of each state. To do so, let us create a table stating the value of each input for each state. For the graph of Fig. 15, the table given in Fig. 16 is constructed as follows:

First, one states (in heavy type) on the table the information which is immediately obvious on the graph. For state c, for example, one can notice P' and Q' which correspond to $I_c \cup O_c'$ and a dash for the variable M which is undetermined in this state. Let us note that establishing this table is a verification of the functionality conditions 1-a) and b) because any variable x_i for one state must be either x_i, or x_i', or undetermined, but not two or three of these simultaneously.

Then, we fill in the remaining spaces corresponding to invariant inputs. For example, Q appears as Q' for state c. Since the value has not changed between b and c, one has also Q' in state b. One can thus follow a chain (or return to the specifications).

Let us determine, with the help of this table, if some states are source states. State c is not a source state because its characteristic monomial is not disjoint from that of d_2.

$$M_c \cdot M_{d_2} = P'Q' \cdot P'Q' = P'Q' \neq 0.$$

On the other hand, a is a source state and $M_a^* = M'P$. Indeed, M' distinguishes a from b and P distinguishes a from c, d_1, and d_2. State b is also a source state with $M_b^* = MP$. Therefore, all the branches which arrive at a and b can be neglected (Fig. 17). State a does not intervene for the realization of the output variables, and besides, it leads to no "useful" state. One can suppress it.

From the graph of Fig. 17 (except state a) we can immediately deduce the wiring of Fig. 18 by applying the wiring rules of Section III-A.

The reader can verify that d_1 is also a source state but this does not give a more interesting solution.

Let us note that the functionality conditions are local for each state; that is, to say, that a global consideration of the graph is not useful. This property allows the treatment of large automata. Examples discussed here are simple because it is not educational to present a more complex example. However, we can cite, among successfully realized networks, the sequential control of a pilot distillation column in the Laboratoire d'Automatique at Grenoble. This sequential system has 16 inputs, 17 outputs, 33 states (33 CUSA), corresponding to about 10 000 total states. It has been realized with 50 integrated circuits

	M	P	Q
a	M′	P	Q′
b	M	P	Q′
c	–	P′	Q′
d_1	–	P′	Q
d_2	–	P′	Q′

Fig. 16. Characteristic monomials for machine M_2.

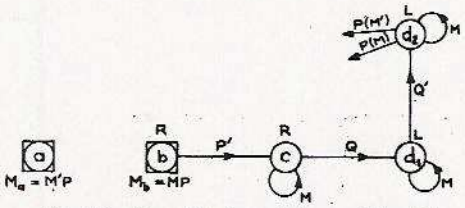

Fig. 17. Simplified graph for machine M_2.

DIL (CUSA plus a small combinational part for inputs and outputs).

F. Multiple-Input Change

We have assumed that multiple input change does not occur. However, let us comment on the possibility of a double-input change and consider two cases.

1) The input variables may change at random. If the change of two input variables appears within a range of 30 ns (with the TTL integrated CUSA) it may have an error. However, if two variables have a mean time between change of 1 s (reasonable for an industrial automaton), their change in the critical range can appear less than once every year. This possibility may generally be neglected compared to other sources of failure.

2) Two-input variables x_1 and x_2 may change simultaneously because they are not independent. Then the system can reach a state which is compatible with $x_1x_2 = 1$ (the graph may eventually be modified). In the example 1 (the graph may eventually be modified). In the example of Fig. 13, a simultaneous change of P' and M', when the system is in state b, leads to state c which is compatible with $P'M'$ (while a is not).

IV. Conclusions

As mentioned in the Introduction, the synthesis method described in the paper is straightforward and systematic; the networks are realized with a single type of cell and testing is not at all difficult.

The necessary material to carry out a network, estimated in the number of TTL packages to be interconnected, is approximatively the same as with any other method. However, the time necessary for completing the synthesis, whether by hand or digitally programmed, is very much reduced.

This method, when taught at both the technicians' and engineers' level, has met with an enthusiastic response from the students.

Several realizations of CUSA have been made via conventional electronics, via medium-scale integration with MOS technology (30 CUSA, with the last metallization mask determining the interconnections), and via integrated circuits with TTL technology. Using the latter, a number of rather important multi-input systems involving hundreds of states and composed of partially independent subsystems have been realized. One such pilot unit controls a distillation column at the Laboratoire d'Automatique at Grenoble.

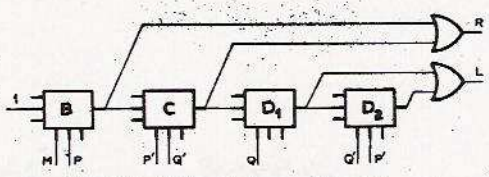

Fig. 18. Network for machine M_2.

Appendix

This Appendix shows an example which is more complex than the examples of the text, but not among the most complicated which have been already realized. It is an 8-input, 4-output, 36-total states, asynchronous system.

Specifications: Let a system have 8 input variables defined as follows: x, $_1x_2$, x_3, u_1, u_2, u_3 are position contacts, as shown in Fig. 19. The input variable M is a starting impulse. The input variable S defines the cycle to be completed. Starting from point A, if $M = 1$, the system realizes either the cycle $ABCDA$ if $S = 1$, or $AEFGABCDA$ if $S = 0$. S can only be changed in the rest initial state. If $M = 1$ at the end of the cycle, a new cycle begins. M can only be released during the last part DA of the trajectory. The 4 outputs H, L, D, R, correspond to the different motions, as shown in Fig. 19.

Synthesis: The description of the corresponding sequential system leads to the functional graph of Fig. 20. The table of characteristic monomials is given in Fig. 21. The states 1,4,7,10 are source states, and we get the network of Fig. 22.

Comments: This example needs 9 D.I.L. integrated circuits (4 SF.C 608, 2 SF.C 607, and 3 other I.C. with inverters and NAND gates for inputs and outputs).

A classical realization with NAND gates has led us to a use of 10 D.I.L. integrated circuit.

This example is quite significant concerning: 1) the amount of material, which is almost the same in the two cases; 2) the time of design which is much shorter with the proposed synthesis (in a ratio from five to ten, from our own experience). Once the functional graph is complete, the synthesis is quite direct.

Acknowledgment

The author wishes to thank Prof. R. Perret for initiating this study, Dr. J. C. Laurent for his contribution, Prof. E. McCluskey and Prof. M. Rabins for their help in writing the final manuscript.

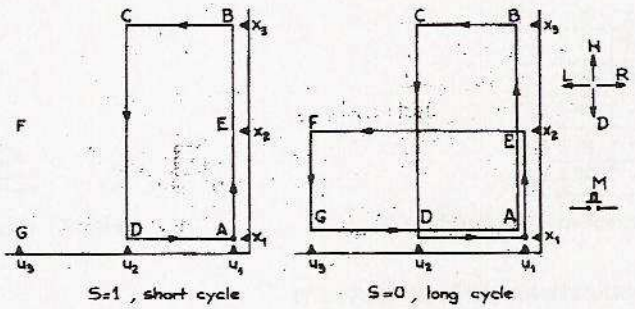

Fig. 19. Cycles to be done.

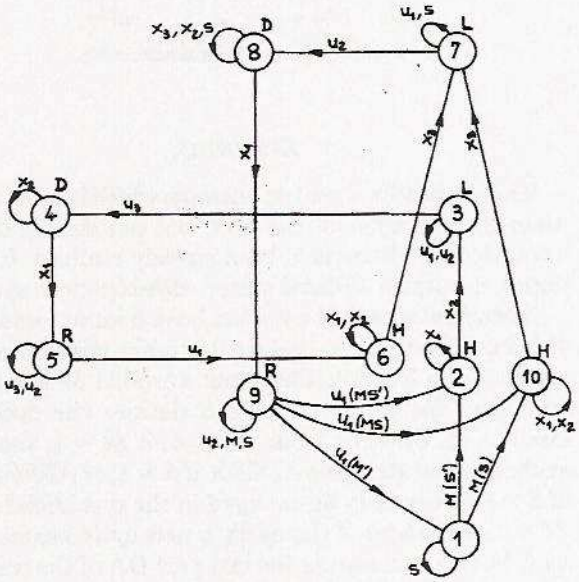

Fig. 20. Functional graph.

	x_1	x_2	x_3	u_1	u_2	u_3	M	S	
1	x_1	x_2'	x_3'	u_1	u_2'	u_3'	M'	—	$M_1^e = u_1 M'$
2	—	x_2'	x_3'	u_1	u_2'	u_3'	M	S'	
3	x_1'	x_2	x_3'	—	—	u_3'	M	S'	
4	x_1'	—	x_3'	u_1'	u_2'	u_3	M	S'	$M_4^e = x_1' u_3$
5	x_1	x_2'	x_3'	u_1'	—	—	M	S'	
6	—	—	x_3'	u_1	u_2'	u_3'	M	S'	
7	x_1'	x_2'	x_3	—	u_2'	u_3'	M	—	$M_7^e = x_3 u_2'$
8	x_1'	—	—	u_1'	u_2	u_3'	M	—	
9	x_1	x_2'	x_3'	u_1'	—	u_3'	—	—	
10	—	—	x_3'	u_1	u_2'	u_3'	M	S	$M_{10}^e = x_3' u_1 MS$

Fig. 21. Table of characteristic monomials.

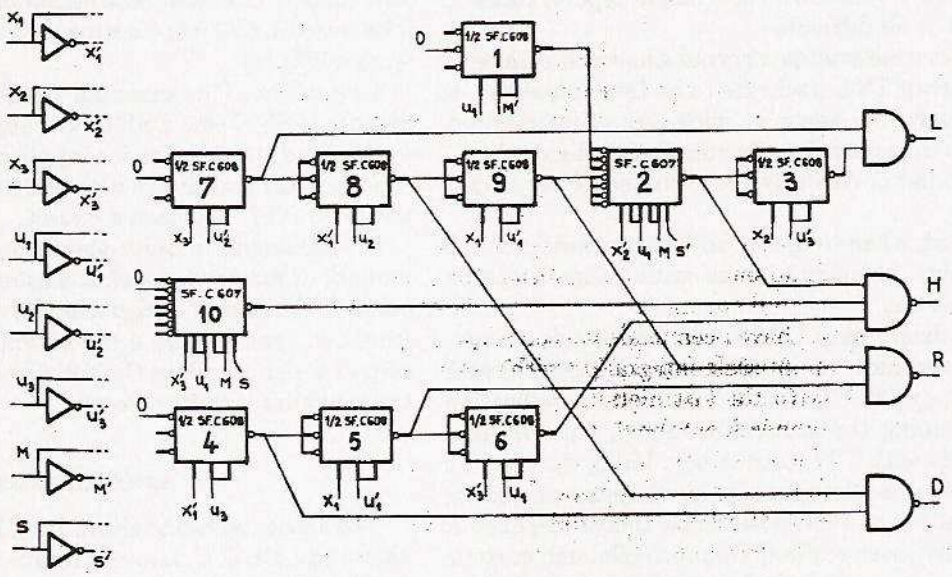

Fig. 22. Network with integrated CUSA (secondary inputs are complemented).

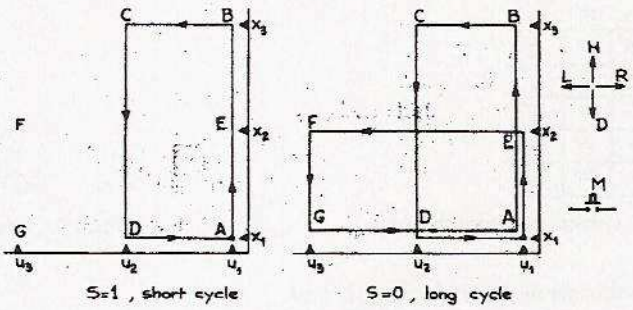

Fig. 19. Cycles to be done.

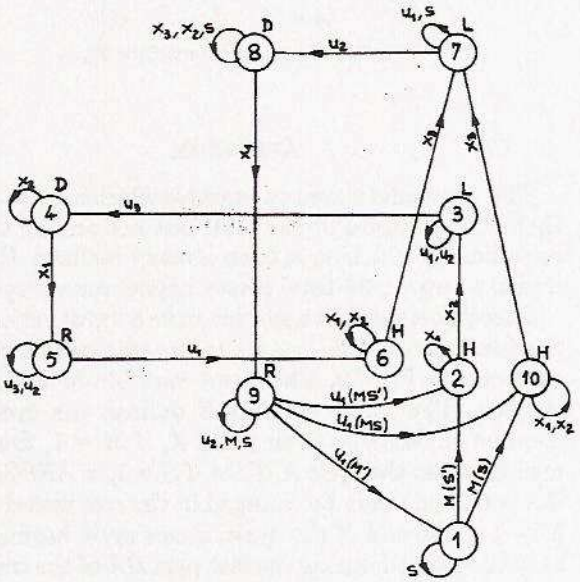

Fig. 20. Functional graph.

	x_1	x_2	x_3	u_1	u_2	u_3	M	S	
1	x_1	x_2'	x_3'	u_4	u_2'	u_3'	M'	—	$M_1^\circ = u_4 M'$
2	—	x_2'	x_3'	u_4	u_2'	u_3'	M	S'	
3	x_1'	x_2	x_3'	—	—	u_3'	M	S'	
4	x_1'	—	x_3'	u_4'	u_2'	u_3	M	S'	$M_4^\circ = x_1' u_3$
5	x_1	x_2'	x_3'	u_1'	—	—	M	S'	
6	—	—	x_3'	u_4	u_2'	u_3'	M	S'	
7	x_1'	x_2'	x_3	—	u_2'	u_3'	M	—	$M_7^\circ = x_3 u_2'$
8	x_1'	—	—	u_4'	u_2	u_3'	M	—	
9	x_1	x_2'	x_3'	u_1'	—	u_3'	—	—	
10	—	—	x_3'	u_4	u_2'	u_3'	M	S	$M_{10}^\circ = x_3' u_4 MS$

Fig. 21. Table of characteristic monomials.

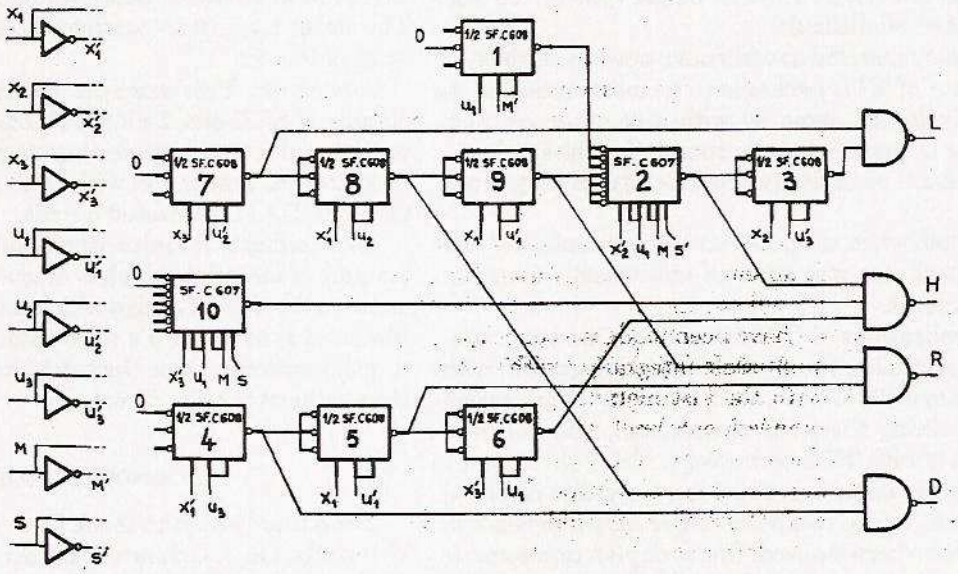

Fig. 22. Network with integrated CUSA (secondary inputs are complemented).

REFERENCES

[1] D. A. Huffman, "The synthesis of sequential switching circuits," *J. Franklin Inst.*, vol 257, no. 3, pp. 161–190; no. 4, pp. 275–303, Mar., Apr. 1954.

[2] D. Ferrari and A. Grasselli, "A cellular structure for sequential networks," presented at the 8th Annual Symposium on Switching and Automata Theory (Austin, TX, Oct. 18–20, 1967).

[3] T. F. Arnold, C. J. Tan, and M. M. Newborn, "Iteratively realized sequential circuits," *IEEE Trans. Comput.*, vol. C-19, pp. 54–66, Jan. 1970.

[4] J. A. Brzozowski and S. Singh, "Definite asynchronous sequential circuits," *IEEE Trans. Comput.*, vol. C-17, pp. 18–32, Jan. 1968.

[5] P. R. Low and G. A. Maley, "Flow table logic," *Proc. IRE*, vol. 49, pp. 221–228, Jan. 1961.

[6] J. Florine and M. Dehagen, "Registres, compteurs et sélecteurs électroniques sans bascules bistables," *Automatisme*, vol. XII, no. 11, Nov. 1967.

[7] A. A. Tal, "Design of static synchronous sequential machines on the basis of standard cells," *Automation and Remote Control*, pp. 2089–2107, Dec. 1966.

[8] R. Perret and R. David, "Synthesis of sequential circuits using basic cell elements," in *9th J.A.C.C.* (Ann Arbor, MI, June 26–28, 1968), pp. 582–596.

[9] R. David, "Réalisation de systèmes séquentiels asynchrones par interconnexion simple de cellules séquentielles identiques," Ph.D. dissertation, Grenoble, 1969.

[10] ——, "Synthèse de réseaux séquentiels cellulaires," *Automatisme*, vol. XV, no. 3, pp. 89–97, Mar. 1970.

[11] J. C. Laurent, "Synthèse cellulaire des automates asynchrones définis par leurs graphes et intégration à grande échelle en technologie M.O.S.," Ph.D. dissertation, Grenoble, 1972.

[12] J. C. Laurent and R. David, "Synthèse cellulaire de systèmes s quentiels complexes définis par leur graphe primitif," *Revu Française d'Automatique d'Informatique et de Recherche Opé ationnelle*, no. J1, pp. 19–34, 1972.

[13] *SESCOSEM Catalogue*, "Logic TTL integrated circuits," (Editio Radio, 9 rue Jacob, 75006 Paris, France), pp. 573–580, 1975.

René David was born in Saint-Nazaire France, on August 8, 1939. He received the engineering degree in 1962 at the Ecole Nationale Supérieure d'Arts et Métiers. In 1963 an accident stopped his work and left him paraplegic. In 1965 he received the Automatic Control Engineering degree at the University of Grenoble and in 1969 he received the Doctorat d'Etat es Sciences Physiques degree at the same university, under the direction of Prof. R. Perret.

Working with the Laboratoire d'Automatique at the Institut National Polytechnique de Grenoble since 1964, he is now Chargé de Recherche at the Centre National de la Recherche Scientifique. His interest lies mainly in synthesis and test of switching circuits, and also in computer structure and traffic simulation.

Dr. David is a member of Association Française pour la Cybernétique Economique et Technique (AFCET).

On Totally Self-Checking Checkers for Separable Codes

MOHAMMAD JAVAD ASHJAEE AND SUDHAKAR M. REDDY, MEMBER, IEEE

***Abstract*—Design of totally self-checking (TSC) checkers for separable codes is studied. Assuming a specific checker design, a sufficient condition on separable codes is derived such that the assumed checker is TSC. It is shown that the proposed checker is applicable to certain Berger codes and residue codes. A class of codes equivalent to Berger codes is derived for which the proposed checker is TSC.**

***Index Terms*—Berger codes, residue codes, separable codes, totally self-checking checkers, unidirectional faults.**

I. INTRODUCTION

SEVERAL researchers have studied the problem of designing totally self-checking (TSC) circuits [1]–[16]. In [3], [4], and [7], TSC checkers for m-out-of-n codes were given. A TSC single-error correcting, and double-error detecting circuit for certain Hamming codes was given earlier [8], [11]. In this paper, we study the problem of designing TSC checkers for separable (or systematic) codes [3], [16], and [19]. This problem was studied earlier (see [15] and [16]), but, as we will show, the results derived therein are applicable only to a special class of separable codes.

We will specifically, propose TSC checkers for Berger codes [3], [17] and some residue codes [18], [19]. The results presented will give the insight necessary to design TSC checkers for all residue codes and for many other separable codes.[1] The motivation for the choice of Berger codes comes from the fact that many large-scale integration (LSI) devices exhibit fault behavior modeled by unidirectional faults [20], [21] and Berger codes are the optimum

Manuscript received October 1, 1975; revised August 2, 1976. This research was supported by NSF Grant ENG72-04042 A02.

The authors are with the Division of Information Engineering, University of Iowa, Iowa City, IA 52242.

[1] For example, for all check sum codes [9], [28].