J. Paul Roth

Diagnosis of Automata Failures: A Calculus and a Method

Abstract: The problem considered is the diagnosis of failures of automata, specifically, failures that manifest themselves as logical malfunctions. A review of previous methods and results is first given. A method termed the "calculus of D-cubes" is then introduced, which allows one to describe and compute the behavior of failing acyclic automata, both internally and externally. An algorithm, called the D-algorithm, is then developed which utilizes this calculus to compute tests to detect failures. First a manual method is presented, by means of an example. Thence, the D-algorithm is precisely described by means of a program written in Iverson notation. Finally, it is shown for the acyclic case in which the automaton is constructed from AND's, NAND's, OR's and NOR's that if a test exists, the D-algorithm will compute such a test.

Introduction

This paper describes a notation and calculus for representing the behavior of failing acyclic automata,* and gives an algorithm—the D-algorithm—for the computation of tests for failures. It is established for acyclic automata (i.e., without feedback) constructed from AND's, OR's, NAND's and NOR's that if a test for any given logical failure exists, the D-algorithm will compute such a test. It is estimated that the algorithm will be about as efficient as the best of previously known techniques, which could not, however, guarantee that a test would be computed even if one existed.

Section 1 begins the treatment by providing a resume of the various methods, including that of the D-algorithm. Section 2 advances the notion and calculus of D-cubes, which are used to describe failure phenomena in circuits. Section 3 describes, via example, a manual procedure for executing the D-algorithm, while Section 4 presents a program, written in the Iverson notation, for the principal procedure of the algorithm and, as well, a verbal elucidation of that program. Section 5 gives a proof of the principal contention that if a test for failure exists, the D-algorithm will find such a test.

We approach the discussion by defining four terms that will be frequently used:

A failure — In a logical circuit (strictly, a Boolean graph, Refs. 2, 3) any transformation of hardware that changes the logical character of the function realized by the hardware.

A primary input — In a logical circuit, a line that is not fed by any other line in that circuit.

A primary output — In a logical circuit, a line whose signal output is accessible to the exterior of the circuit.

A test (for a failure) — A pattern of signals on primary inputs such that the value of the signal on some primary output will differ according to the presence or absence of that failure. Also, *test* will often be extended to mean the total pattern of signals on all lines of the circuit.

1. Review of methods

- *The truth-table method.*

In this method and others, let G denote the function described by the circuit. If it is a multiple output circuit, G might be thought of as several Boolean functions, one for each output. Let F denote the function described by the circuit with a given failure. To find a test to determine whether or not this particular failure has occurred, it is "merely" necessary to compare the truth-tables for these two functions. This, of course, requires that the truth tables for each function actually be constructed and then compared to ascertain those inputs for which the output differed. Presumably, this process would be necessary for every failure of interest. Then, to determine a subset of these tests to ascertain whether or not any failure occurs, one might go through some sort of covering procedure

* References 1a & 1b show techniques such as SCAN for adapting such methods to circuits with feedback.

(*e.g.*, the extraction algorithm, Ref. 4). Clearly, this method would be effective only for small problems, certainly not for problems in 100 variables.

The method by complements

Suppose that we compute the function G, which is defined by the good circuit and $\bar{G}$, the complement of G (the set of complements, if there are many outputs). This computation may be done, *e.g.*, by the π^*-algorithm (Ref. 5). This computation gives, for each output, a normal form expression for the function and its inverse. Then to determine the set of all tests to detect the given failure, where F again denotes the function of the failing circuit, it would be necessary only to form the intersections $\bar{G} \cap F$ and $G \cap \bar{F}$. This method is superior to the truth-table method in that it works with normal form expressions rather than with canonical terms. However, the formation of the intersection can be a formidable problem; for instance, if F and G each were expressed by a thousand terms (a thousand cubes), then a million intersections would be required. Notwithstanding, the method is in many cases a substantial improvement over the truth-table method. A procedure similar to that for the truth-table method might be used to find a set of tests to detect whether a failure has occurred and a second set of tests to ascertain, knowing that a failure has occurred, just which it is.

Pruning

This method is described in Reference 5. Its basic innovation is that in one step it accomplishes the process of computing F, $\bar{F}$, G and $\bar{G}$ and forming their intersections, as described above, in roughly the same amount of computation as to form F alone. The method amounts to the following: At the point in the circuit where the failure would occur, it is "cut" and the π^* procedure is used to compute the output in terms of the standard inputs and the "pseudo-input" at the point of cut. That is to say, the circuit is replaced by another which has the point of cut as an input (the branch above the point of cut is discarded). The branch remaining after "pruning" is then substituted, in the arrays produced, for the pseudo-input, to obtain a set of all tests to detect the given failure. Pruning is considerably more efficient than the method by complements, yet it frequently shares with the latter a need for a large memory and, possibly, a very large amount of computing time, for the so-called ON-arrays and OFF-arrays generated can be very large. The program of a considerably refined version of this algorithm, written by P. N. Sholtz, J. L. Sanborn, and J. M. Galey, has seen interesting use in IBM.

The effective computability of the pruning algorithm is limited chiefly by the size of its storage requirements for problems with a large number of inputs. For example, it was calculated that one problem in 115 variables would require 10^8 reels of IBM magnetic tape to record a minimum normal form expression for the complement of the function, yet the circuit itself comprised only 65 logical blocks.

Tracing

This method has been developed and brought to a high point of efficiency by C. B. Stieglitz.* A failure at a given point in the circuit is assumed. Then one computes the signals that are necessary at that point for the failure to be detected on the output of the block to which the point is immediately connected. The signals are "traced" to an output assigning "as-you-go," on lines feeding each block encountered, signals that are required for transmitting information concerning the alleged failure through the block. This tracing proceeds systematically, with backing up taking place whenever "conflicts" occur between signals required to appear on any given line, and continues until an output is found that is sensitive to changes in the signal of the failing line. When one such output is found, the tracing proceeds backwards from the point of failure to the inputs, in hope of finding a set of inputs which will bring up a set of signals on the failing line and on other lines interior to the circuit in order to effect this test. This procedure is continued until such a set of inputs is obtained or a specified running time is exceeded. Thus, this process computes a hypothetical test for each failure, after which it is necessary to employ "simulation" to determine whether or not the hypothetical test is a true test for that failure. If not, then computation and simulation must continue. The advantage of the tracing method is that more often than not it is fast. The disadvantage, as simple examples show, is that there is no guarantee that in fact a test will be computed, even though a test may exist.

The tracing method was improved and adapted for FLT (Fault Location Technology) by K. Maling, F. N. Evans, M. W. Evans, and W. C. Carter (Ref. 1a) and R. J. Preiss and embodied in an IBM 7094 program. This program, joined with the SCAN concept (Ref. 1b) has been massively used for the computation of diagnostic tests for IBM System/360.

The D-algorithm

The D-algorithm we are to describe proposes a new calculus of complexes in which the internal line structure of a circuit is utilized to describe the function and its failures. (The notation it employs was alluded to in Reference 7.) The method has the advantage that the number of "terms" (more precisely, cubes) that are required to describe the function of a circuit and its failures is directly and solely proportional to the number of logical blocks in the circuit. It will be seen that if a test exists for a given failure, the algorithm will find such a test. In a particularly simple manner, the theory is extended to include the detection of

*A paper describing this method is in preparation. See Ref. 6.

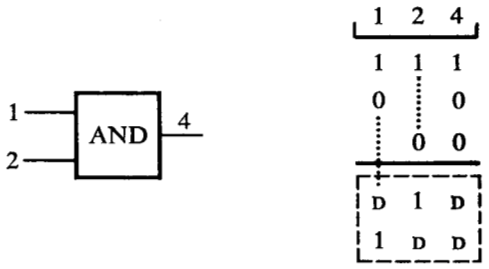


Figure 1 An AND circuit, its singular cover and, within dashed lines, two D-cubes.

short circuits between logically distant lines as well as *any* logical failures associated with the logical block.

Although explicit comparisons of methods must await results from programs soon to be run (on an experimental Iverson time-sharing interpreter at the IBM Research Center), it can be supposed that the running time of the D-algorithm will not be greater than that of the tracing method. Indeed, it should usually be considerably less, for the reason that computation of a true test with the D-algorithm requires about the same running time as does the computation of a single hypothetical test via the tracing method. Memory requirements for the D-algorithm are expected to be comparable to those for the tracing method.

2. Calculus of D-cubes

To describe D-cubes and their properties can best be done by presenting a few examples: Figure 1 consists of a single AND block with inputs 1 and 2, and with output 4. At the right is a kind of truth table, the first row specifying that the output is 1 when both inputs are 1, the second and third, that the output is 0 when either input is 0. This is termed the *singular cover* of the function (Ref. 2). Now, 1 and 2 are the *input coordinates* and 4 is the *output coordinate* of the logical block. The cubes that make up this singular cover may also be written with x's in lieu of blanks, which would be a form closer to the author's earlier notation. Employing x's we have

1	2	4
1	1	1
0	x	0
x	0	0.

The cube 0 x 0 stands for the two vertices

1	2	4
0	0	0

and

1	2	4
0	1	0

or, in a notation imposed by typography, the two vertices $^{1}0\ ^{2}0\ ^{4}0$ and $^{1}0\ ^{2}1\ ^{4}0$. In any program of the method, the blanks or, equivalently, the x's, will of necessity be represented in the coding of symbols.

Now, Figure 1 also gives two so-called D-cubes,

1	2	4
D	1	D

and

1	2	4
1	D	D

or, in the alternative notation, $^{1}D\ ^{2}1\ ^{4}D$ and $^{1}1\ ^{2}D\ ^{4}D$. These have the following interpretation: The letter D may assume just two values, 0 and 1, so that D 1 D stands for the two states $^{1}1\ ^{2}1\ ^{4}1$ and $^{1}0\ ^{2}1\ ^{4}0$; it specifies that, when the logic block is working properly, the value of the signal on the output line 4 must be the same as that of the input on line 1, provided that line 2 is kept at 1.

Recalling the definition for a test, we see that, for example, the cube $^{1}D\ ^{2}1\ ^{4}D$ contains two embryonic* tests: The input signal 1 1 1 (obtained by setting D = 1) constitutes a test for line 1 stuck-at-0 and line 4 stuck-at-0; the input signal 0 1 0 (for D = 0) is a test for each of these lines stuck-at-1. A similar interpretation holds for D-cube 1 D D. In order to be quite clear to the reader interested in making comparisons with the author's previous notation, we consider another example. First, however, we remark that a vertex in the 6-dimensional "space" of coordinates 1, 2, 3, 4, 5, 6 is simply a vector of six binary digits, for example,

1	2	3	4	5	6
1	0	0	1	0	1.

In the three-dimensional cube $^{1}1\ ^{2}x\ ^{3}x\ ^{4}1\ ^{5}0\ ^{6}x$, each x is able to take on, independently of any other x, two values; thus this cube "covers" 2^3 vertices formed by allowing each x to assume values 1 or 0. Likewise, the cube $^{1}D\ ^{2}1\ ^{3}D\ ^{4}0\ ^{5}D\ ^{6}D$ represents only *two* vertices, since all D's are to be thought of as *all* having the same values together. We shall be dealing with "mixed" cubes, consisting of cubes containing *both* D's and x's (or, equivalently, blank spaces rather than x's).

For example, if we hook up three logic blocks as in Figure 2, the behavior of the block with output 4 of Figure 1 is given by the first three cubes of the cover. (The cube $^{1}1\ ^{2}1\ ^{3}\ ^{4}1\ ^{5}\ ^{6}$ may also be written $^{1}1\ ^{2}1\ ^{3}x\ ^{4}1\ ^{5}x\ ^{6}x$, and similarly for the other cubes.) The next three cubes specify the input-output relationship of block 5, and the next three specify that of block 6. These nine cubes are termed the *singular cover* of the Boolean function of line 6, the output signal, in terms of its primary input signals. Internal lines are utilized in making explicit this relationship and in general, particularly for large circuits, this mode of expression is phenomenally more concise than an ordinary cubical cover (normal form). Furthermore, as we shall see, it is particularly well suited for computing tests for failures of the circuit.

* We use the term "embryonic" to emphasize that the test pattern refers only to the inputs and outputs of the block itself.

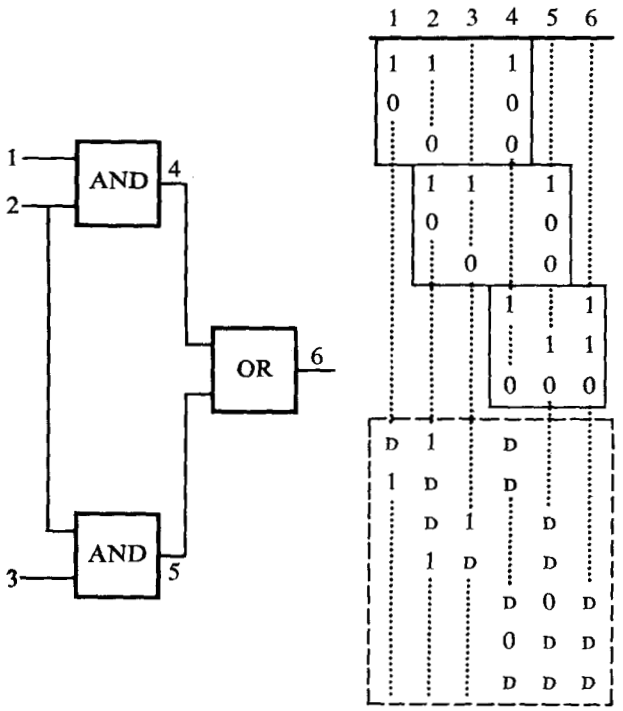


Figure 2 A simple circuit with its singular cover and, within dashed lines, D-cubes.

Below this singular cover are listed the "primitive" D-cubes for the circuit. Each of these contain two embryonic tests for specific failures on each block. These embryonic tests are in terms of signals on the immediate outputs of each block. For example, the cube ${}^4\mathrm{D}\ {}^5\mathrm{0}\ {}^6\mathrm{D}$ contains two tests, ${}^4\mathrm{1}\ {}^5\mathrm{0}\ {}^6\mathrm{1}$ and ${}^4\mathrm{0}\ {}^5\mathrm{0}\ {}^6\mathrm{0}$; these are tests for failures of line 4 in terms of the output on line 6. The seven D-cubes listed constitute all the D-cubes necessary to compute tests for the entire circuit. (This reflects the general case in that such a body of primitive D-cubes of logical blocks, together with the primitive D-cubes of the failures (see page 282), and together with the "multiple D-cubes" which are computed on a demand basis, are sufficient to compute tests for all failures.)

Now let us see how we can *combine* these D-cubes to compute tests for lines in terms of signals imposed on primary inputs. For example, the D-cube ${}^1\mathrm{D}{}^2\mathrm{1}{}^4\mathrm{D}$ contains tests for line 1 in terms of line 4; the D-cube ${}^4\mathrm{D}\ {}^5\mathrm{0}\ {}^6\mathrm{D}$ contains tests for 4 in terms of 6. When these are combined or "intersected" they yield

1	2	4	5	6
D	1	D	0	D,

and we now have tests for 1 *and* 4 in terms of 6. Now, the D-cube above has this interpretation: When line 2 has signal 1 and line 5 has signal 0, lines 1, 4, and 6 will have the same signal D, with D equal to 0 or 1. Thus, for example, to test for whether line 4 is stuck-at-1 (Cf. Reference 5) it would be sufficient to have signals 1 and 0 on lines 2 and 5, respectively, and to have 0 on line 1; then if 4 is stuck-at-1, 6 will be 1, but otherwise 6 will be 0.

If we let some circuit have n lines, then a D-cube of that circuit is an n-tuple of symbols consisting of 1's, 0's, D's, $\bar{\mathrm{D}}$'s, X's, or blanks. The D-cube is termed *complete* if no coordinate is X or blank. The interpretation of a complete D-cube is fairly simple: each 1 or 0 represents a signal on the corresponding line; each D indicates a possible pair of signals, either 1 or 0 but both being the same for all lines corresponding to coordinates having the value D; conversely, all lines having coordinates $\bar{\mathrm{D}}$ have the same value but one that is opposite to that of the lines having coordinates D. If a D-cube is not complete, then it has coordinates which are not specified; i.e., are blank. Let there be r of these. Then this D-cube represents 2^{r+1} possible configurations of signals on the circuit. As indicated above, these blanks may be changed to X's as for the standard cubes that have been treated by the author. Thus in Figure 2, the incomplete D-cube ${}^1\mathrm{D}\ {}^2\mathrm{1}\ {}^3\ {}^4\mathrm{D}\ {}^5\mathrm{0}\ {}^6\mathrm{D}$ may be written ${}^1\mathrm{D}\ {}^2\mathrm{1}\ {}^3\mathrm{X}\ {}^4\mathrm{D}\ {}^5\mathrm{0}\ {}^6\mathrm{D}$.

The *vertices* of a D-cube having r X's are obtained by setting each X to a 0 or 1, each D to a 1, and each $\bar{\mathrm{D}}$ to a 0 (or vice versa, each D to a 0, and each $\bar{\mathrm{D}}$ to a 1) to define 2^{r+1} such vertices. We say D-cube A D-*contains* D-cube B if the set of vertices of D-cube A contains all those of D-cube B. For example, A=1 X 0 X D $\bar{\mathrm{D}}$ must D-contain 1 1 0 x 0 1 and 1 x 0 1 0 1, as well as six others.

Construction of $c(T, F)$

Let T be a test and F be a logical failure in a circuit. Then each test T and failure F define a D-cube, $c(T, F)$, in the following manner: If, for any coordinate, the signal as induced by the test input in line i is the same, say 1, whether or not a failure exists, then the D-cube has this signal, say 1, as its i^{th} coordinate. If the perfect circuit has a 1 on a given line in response to the given test and the circuit with the given failure has a 0, then assign to the corresponding coordinate a D. If the reverse, assign a $\bar{\mathrm{D}}$.

Observe that this construction is possible only if the logic circuit is a Boolean graph, i.e., has no feedback (for if not, the signals on lines in the failing circuit are not unique). Note also that $c(T, F)$ conversely defines T, being the values of $c(T, F)$ on the primary inputs pi.

Lemma 1: If a given test T detects a given logical failure F, then the coordinates expressed by the corresponding D-cube, $c = c(T, F)$, defined by T and F, contain a connected chain of coordinates having values D or $\bar{\mathrm{D}}$, linking the output line of the failing block to a primary output.

Proof: By definition, some primary output is a function

of the failure in the sense that its value changes depending upon the presence or absence of the failure. This dependence must be through some chain connecting the failing block to the primary output. For if there were no connected chain of coordinates having the value D or $\bar{D}$ linking these two, then, along every path between the failing block and the sensitive primary output, the chain of D's would terminate; the "next" block in the chain would have a fixed output value whether or not the failure had occurred. Thus, along each path from failing block to output, eventually there would be no change due to the failure. Consequently, the primary output would also not change. *Q.E.D.*

The general objective is now to combine or "intersect" the "primitive" D-cubes constructed from each block to achieve a test for a corresponding set of failures.

• *Intersection of cubes*

Let the coordinates $(a_1, \cdots, a_n)$ and the coordinates $(b_1, \cdots, b_n)$ specify cubes a and b, where a_i and b_i are 0 or 1 or x. Cubes a and b will intersect *in an empty cube* ϕ, if, for some i, $a_i = 1$ and $b_i = 0$, or $a_i = 0$ and $b_i = 1$. If this condition does not occur their intersection is specified by coordinates $(c_1, \cdots, c_n)$, where each $c_i = a_i \cap b_i$ is formed according to the rules below (Cf. Ref. 8):

$0 \cap 0 = 0 \cap x = x \cap 0 = 0;$

$1 \cap 1 = 1 \cap x = x \cap 1 = 1;$

$x \cap x = x.$

For example, $0\ x\ x \cap 1\ x\ x$ intersects in the empty cube ϕ, but

$0\ 1\ x\ x\ 1 \cap 0\ x\ 1\ x\ 1 = 0\ 1\ 1\ x\ 1$

and

$x\ x\ x\ 1 \cap 1\ x\ 0\ x = 1\ x\ 0\ 1.$

• *Primitive D-cubes of a failure.*

Consider a logical block with a given singular cover specifying $\alpha 1$, the totality of conditions under which the output of the block is 1 (these are specified by all those cubes whose output coordinate is 1) and $\alpha 0$, the totality of conditions under which the output is 0 (these are specified by all those cubes for which the output is 0). Now let F be a failure of the logical block that changes the logical function which it performs. Let the singular cubes for which the output of the failing block is 1 be denoted by $\beta 1$ and those for which the output is 0, by $\beta 0$. For example, suppose the logical block is an AND which under a given failure is transformed into an OR, as shown below, with 1, 2 as input coordinates and 4 as output coordinate.

1	2	4	
1	1	1	$\alpha 1$
0	x	0	$\alpha 0$
x	0	0	$\alpha 0$
1	x	1	$\beta 1$
x	1	1	$\beta 1$
0	0	0	$\beta 0$

Then the totality of input conditions for which the outputs disagree, depending on whether or not there is a failure, is obtained by: (1) changing the output coordinate of all cubes of $\alpha 1$ and intersecting them with $\beta 0$ and (2) changing the output coordinate of all cubes of $\alpha 0$ and intersecting them with $\beta 1$. Those cubes in the first class we shall denote by assigning a D to their output coordinate and those of the second, by a $\bar{D}$.

In the example, we thus generate the D-cubes

1	2	4
0	1	$\bar{D}$
1	0	$\bar{D}$

These cubes now have the following interpretation: When the inputs 0, 1 are applied respectively to lines 1 and 2, the output will be 0 if the circuit is perfect, or 1 if it has changed to an OR output. Similarly for the input 1, 0.

In the case of a multiple output block such as we shall meet for the diagnosis of shorts between lines, the procedure for generating the D-cubes is exactly similar. For example, let 1, 2, 3, 4 be input coordinates and 5, 6 output coordinates. Suppose that the singular cover for the perfect circuit contains the cube $^1 1\ ^2 x\ ^3 1\ ^4 x\ ^5 1\ ^6 0$, and the failing circuit contains $^1 x\ ^2 1\ ^3 x\ ^4 1\ ^5 0\ ^6 1$. Then the construction yields the D-cube $^1 1\ ^2 1\ ^3 1\ ^4 1\ ^5 D\ ^6 \bar{D}$.

• *Primitive D-cubes of a logical block.*

For purposes of computing tests for failures which can be detected by primary outputs of the circuit, we introduce another type of D-cube, one which essentially specifies the signals on all inputs to a block but one (or more) under which a change of signal in this input (inputs) induces a change on the output of the block. The example of Figure 1 depicts these D-cubes.

We shall first give the construction for change on a single input coordinate i. For each cube in the singular cover for which the i^{th} coordinate is not x, change its value and change the value of the output coordinate. Intersect this cube with each other cube in the singular cover; for each such cube in the intersection, assign the value D to the i^{th} coordinate; if the value of the output in the original cube is the same as that of the i^{th} coordinate, assign to it the value D; otherwise, $\bar{D}$.

Basically, we need two kinds of primitive D-cubes: D-cubes with only one input coordinate equal to D or $\bar{D}$, and multiple input D-cubes, which have more. In constructing the multiple input D-cubes, it is expedient to deal with canonical covers, i.e., covers of cubes containing no X's. Otherwise, the procedure for changes in r input coordinates is exactly analogous to that for one.

For example, if the singular cover contained the cubes

1	2	3	4	5	6
1	0	1	0	X	1
0	1	X	0	1	0 ,

with 1, 2, 3, 4, and 5 the input coordinates and 6 the output coordinate, and with 1 and 2 the particular coordinates, the D-cube

1	2	3	4	5	6
D	$\bar{D}$	1	0	1	D

would be constructed. This has the interpretation that, when input lines 3, 4, and 5 have values 1, 0, and 1, then when 1 and 2 change respectively from 10 to 01 the output on line 6 changes from 1 to 0 and vice versa.

Lemma 2. The totality of all input configurations, such that an inversion in the value of each r particular input lines effects a change in the output, is obtained by the above construction and each such input configuration is contained in one of these primitive D-cubes.

Proof: This lemma follows directly from the construction of these primitive D-cubes: For each cube in the cover, examine the particular r coordinates in question. In canonical terms, every possible input configuration is represented. Suppose there was a multiple input D-cube for which there are r input D-coordinates. Then, setting D = 1, we are yielded one term which must appear as a canonical term in the cover; similarly the cube obtained by setting D = 0 also must be in the cover and must by construction be obtained from the first by inversion of the particular r input coordinates. Hence, this D-cube would be obtained by construction from the canonical cover. *Q.E.D.*

• D-*intersection.*

First we define the coordinate D-intersection of the symbols 0, 1, X, D, and $\bar{D}$. This requires the introduction of four new symbols: ϕ, meaning empty D-intersection; ψ, meaning that D-intersection is not defined; and λ and μ, signifying a more complicated subroutine for the definition shown in the table immediately below:

$\cap$	0	1	X	D	$\bar{D}$
0	0	ϕ	0	ψ	ψ
1	ϕ	1	1	ψ	ψ
X	0	1	X	D	$\bar{D}$
D	ψ	ψ	D	μ	λ
$\bar{D}$	ψ	ψ	$\bar{D}$	λ	μ .

Thus, if for any coordinate, the coordinate D-intersection is ϕ, then the D-intersection is said to be the empty cube. If any coordinate intersection is ψ then the D-intersection is undefined.

Assume now in what follows that no coordinate D-intersection ϕ or ψ occurs. Then if *both* λ and μ occur, the coordinate D-intersection is not defined. If only μ occurs, then for these coordinates let $D \cap D = D$, $\bar{D} \cap \bar{D} = \bar{D}$ and let the D-intersection be the set of coordinate intersections. If only λ occurs, then in the second factor change all coordinates which are D to $\bar{D}$, and which are $\bar{D}$ to D. Then using the rules for coordinate D-intersection of $D \cap D = D$, $\bar{D} \cap \bar{D} = \bar{D}$, the D-intersection is the set of these coordinate D-intersections. It is seen, by comparing the intersection table in Reference 8 with that above, that when cubes a and b are ordinary (nonsingular) cubes, i.e., with no coordinates equal to D or $\bar{D}$, that intersection and D-intersection coincide.

Lemma 3: Where products are defined, D-intersection is commutative and associative.

In the algorithm to follow, we will use the *restricted* D-intersection which insists that there be at least one coordinate for which both cubes have the value D or $\bar{D}$; that is, their D's must "interact." We shall say that D-cube a D-*contains* D-cube c if for each coordinate i, $a_i = c_i$ or X. It follows from this definition that if a D-contains c, then the set of all vertices of a contains the set of all vertices of c, so that a also contains c. If a D-contains c we shall write $a \supset c$.

Lemma 4. Let T be a test for a failure F of logic block λ and again let $c(T, F)$ be the D-cube defined by this test and failure. Then $c(T, F)$ is D-contained in a primitive D-cube of failure F.

Proof: The coordinate(s) of the output of λ in $c(T, F)$ must clearly be D or $\bar{D}$ since otherwise no primary output would be capable of distinguishing the existence or nonexistence of failure F, by the above lemma. Thus consider the D-cube defined by having the same values as $c(T, F)$ for the immediate inputs to λ, the same value D (or $\bar{D}$) which $c(T, F)$ has for the output of λ and, for all other coordinates, its value is X: this is clearly a primitive D-

cube of the failure F and it clearly D-contains $c(T, F)$. *Q.E.D.*

Lemma 5: If D-cubes a and b each D-contain D-cube $c \neq \phi$, then their D-intersection is defined and it D-contains c.

Proof: By hypothesis, for each coordinate, $a_i = c_i$ or x and $b_i = c_i$ or x, so that $a_i \cap b_i = c_i$ or x or μ. Now if the coordinate D-intersection is μ, then c_i must be D or $\bar{D}$ and, according to the rule for D-intersection, this coordinate is changed to D or $\bar{D}$. It follows for this case that $a_i \cap b_i = c_i$. Hence in general, the coordinate D-intersection $a_i \cap b_i = c_i$ or x so that $a \cap b$ by definition D-contains c. *Q.E.D.*

Lemma 6: Let T be a test for a failure F of a given Boolean graph (a logical circuit without feedback). Let $c(T, F)$ denote the D-cube defined, as described above, by T and F. Let λ be a logic block, not the site of the failure F, for which the output coordinate of λ in $c(T, F)$ is D (or $\bar{D}$). Then $c(T, F)$ is D-contained in a primitive D-cube of λ.

Proof: If an output coordinate of block λ of $c(T, F)$ consists of a D or $\bar{D}$, then at least one input coordinate to the block of $c(T, F)$ must also be D or $\bar{D}$, for if all the input coordinates had values 1 or 0, i.e., did not change whether or not failure F existed, then the output also obviously could not change.

Let $a_{i(1)}, \cdots, a_{i(r)}, b_{j(1)}, \cdots, b_{j(s)}$ be the coordinates of $c(T, F)$ for logic block λ, with the a's input coordinates and the b's the output coordinates. Assigning to D the value 1 and to $\bar{D}$ the value 0, we define a cube which is "covered" by the singular cover of λ; i.e., each of its vertices is contained in some cube of the singular cover. Similarly a cube is defined for the opposite assignment: D = 0 and $\bar{D}$ = 1. On the other hand, these two cubes (they are vertices of the cover of λ), when combined by the method above, go together to form a primitive D-cube *pdc* of the block λ. Now, to be explicit, if *pdc* is extended over all coordinates of the Boolean graph by assigning the value x to those as yet unassigned, then it is clear that *pdc* D-contains $c(T, F)$. *Q.E.D.*

• *Detection of short circuits.*

We next consider shorts between lines of the circuit. Consider the typical case as shown in Fig. 3, consisting of logical blocks A and B whose output lines a and b cross. If the lines short at the point of crossing we shall assume that the short behaves as a so-called dot-OR function. The treatment is exactly analogous in case the short behaves as an AND. We assume, therefore, the existence of a pseudo-block C with inputs a and b and outputs a^* and b^*. When no short occurs $a^* = a$ and $b^* = b$. Under failure, however, a^* and b^* become the OR of a and b: $a^* = b^* = a \vee b$. It is easy to show therefore that the D-cubes which

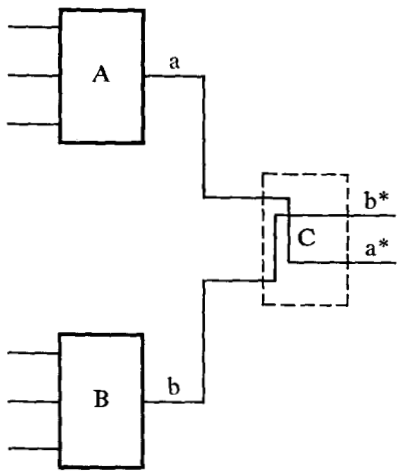


Figure 3 Shorts between lines: an illustrative circuit.

contain all tests to detect these failures are

a	b	a^*	b^*
1	0	1	$\bar{D}$
0	1	$\bar{D}$	1.

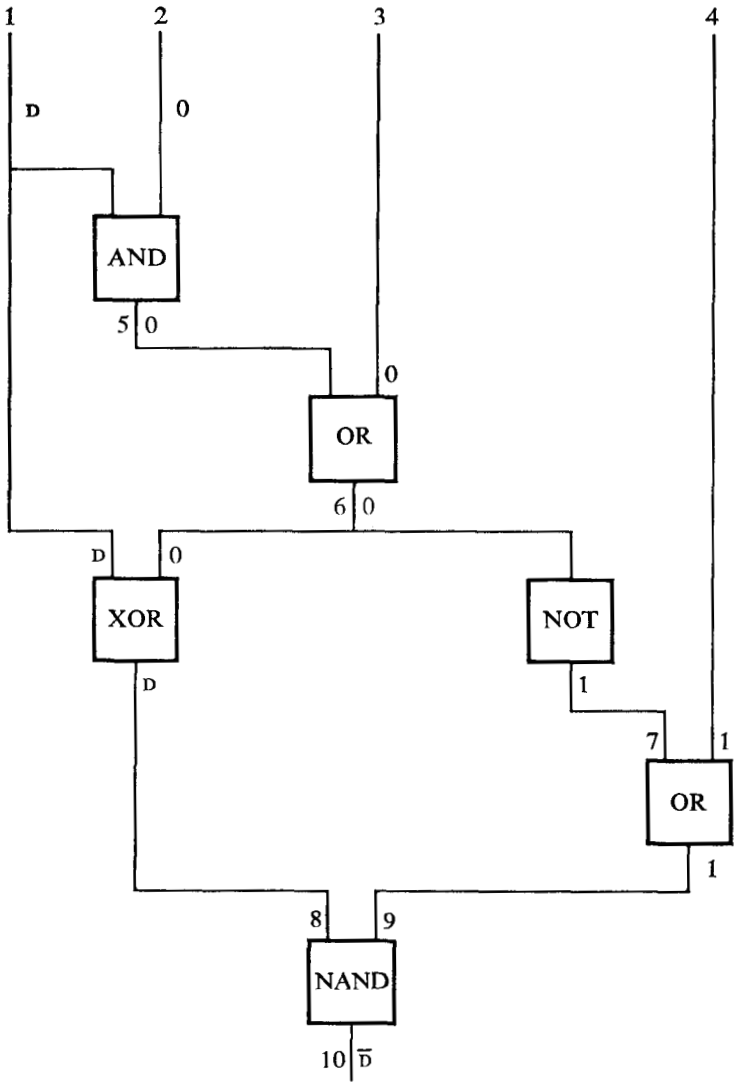
These cubes may therefore be inserted into the algorithm for the generation of tests of Section 3 or 4, to obtain tests in terms of primary inputs for this failure.

3. Manual procedure for D-algorithm

The D-algorithm will be described in two forms: First by means of a manual procedure in this section, in which the set of primitive D-cubes will be priorly computed and stored, and second, by means of a program written in Iverson notation in Section 4. In both cases the underlying steps are quite simple: *First*, having selected a primitive D-cube of a given failure, this cube is successively D-intersected with primitive D-cubes in an attempt to form a connected chain of D-coordinates to a primary output; *second*, having generated such a D-cube, we attempt to "complete" it by means of D-intersecting it with the singular cover S. This will be called the CONSISTENCY operation.

We shall describe the manual procedure by means of the example shown in Figure 4. The figure on the left is described analytically by the singular cover S on the right, consisting of rows a through r. The primitive D-cubes of the logic blocks (the single-output variety) are represented by the rows A through M.

The first step in the algorithm is to select at the site of the failure a primitive D-cube of the failure. Consider first the case of a failure in line 1 (we treat the failures stuck-at-1 and stuck-at-0 simultaneously): Thus the initial test cube $tc^0 = {}^1D$. We form the following chain of D-inter-



	1	2	3	4	5	6	7	8	9	10
a	1	1			1					
b	0				0					
c		0			0					
d			1			1				
e					1	1				
f			0		0	0				
g						1	0			
h						0	1			
i	1					0		1		
j	0					1		1		
k	0					0		0		
l	1					1		0		
m				1					1	
n							1		1	
o				0			0		0	
p								1	1	0
q								0		1
r									0	1
A	D	1			D					
B	1	D			D					
C			D		0	D				
D			0		D	D				
E						D	D̄			
F	D					0		D		
G	D					1		D̄		
H	0					D		D		
I	1					D		D̄		
J				D			0		D	
K				0			D		D	
L								D	1	D̄
M								1	D	D̄

Figure 4 The circuit, singular cover, and primitive D-cubes for the D-algorithm of Section 3.

sections: (The order for D-intersection should be lexicographical; it turns out, in this example, that the first such intersection that "works" is the one shown.)

$$tc^1 = tc^0 \cap F = \underset{1\ 6\ 8}{\text{D 0 D}}$$

$$tc^2 = tc^1 \cap L$$

$$= \underset{1\ 6\ 8}{\text{D 0 D}} \cap \underset{8\ 9\ 10}{\text{D 1 D̄}} = \underset{1\ 6\ 8\ 9\ 10}{\text{D 0 D 1 D̄}}.$$

Now for each D-cube tc we define the *activity vector* a consisting of the set of all coordinate numbers j of tc for which: (1) the coordinate tc_j = D or D̄; and (2) this coordinate j is a primary output of the subgraph of the circuit defined by those coordinates of tc not equal to x. In the example above a^1 of tc^1 is $\{1, 8\}$; for tc^2, $a^2 = \{10\}$.

In Section 4, the notion of the D-fanout of a, fa, figures prominently. It keeps track of the number of successors of each element of a which are potential extenders of the D-chain to primary outputs, i.e., are on the "frontiers" of tc; if a line is a primary output, then 1 is added to this

number (see step 3 of Fig. 5.). In this example $fa^1 \equiv ({}^1 1, {}^8 1)$.

Thus, returning to the description of the algorithm, when the D's have been driven forward as far as possible and primary outputs *po* have been encountered, the CONSISTENCY operation, moving back to the primary inputs, via D-intersecting with S, is begun. Thus, in the example, since the activity vector a^2 of tc^2 consists only of the primary output 10, the first part of the algorithm is finished.

The next step is the CONSISTENCY operation. The purpose of this operation is to "fill in" the remaining coordinates and to ensure that they are "consistent" with the singular cover. In general, it is necessary to have "prime" cubes (prime implicants) as a singular cover for this operation. This time we shall, for the sake of convenience, write down all the singular cubes in tabular fashion and then intersect them all simultaneously with tc^2.

	1	2	3	4	5	6	7	8	9	10
$tc^2 =$	D					0		D	1	$\bar{D}$
n							1		1	
h						0	1			
f			0		0	0				
$\cap c$		0			0					
tc^5	D	0	0		0	0	1	D	1	$\bar{D}$

The fourth coordinate is blank (or x); this means that it may be given an arbitrary value. Thus tc^5 describes four tests, the primary inputs for which are:

1	2	3	4
1	0	0	0
1	0	0	1
0	0	0	0
0	0	0	1 .

The first two of the four tests are for line 1 being stuck-at-0, the last two, for line 1 stuck-at-1.

Another path in the algorithm also applied to the D-cube ${}^1 D$ of the failure line 1 stuck-at-1 (or stuck-at-0) is the following, shown up to tc^3:

	1	2	3	4	5	6	7	8	9	10
$tc^0 =$	D									
$tc^1 = tc^0 \cap A =$	D	1			D					
$tc^2 = tc^1 \cap D =$	D	1	0		D	D				
$tc^3 = tc^2 \cap E =$	D	1	0		D	D	$\bar{D}$			

At this point, the vector a of active coordinates consists of 1, 6, and 7. Now the block with output 8 has both 1 and 6 as inputs and, therefore, the need for a "double" D-cube is indicated. Let us go through this computation.

The first cube in the singular cover for block with output 8 is ${}^1 1\ {}^6 0\ {}^8 1$. In accordance with the method for generating multiple D-cubes (Section 2), change the input coordinates affected, 1 and 6, and its output, 8, to obtain ${}^1 0\ {}^6 1\ {}^8 0$ and D-intersect it with the remaining cubes ${}^1 1\ {}^6 1\ {}^8 0$ and ${}^1 0\ {}^6 0\ {}^8 0$. They do not D-intersect so that a double D-cube does not exist and the generation of the test cube tc^3 is abandoned.

It is instructive to compute a test for line 2. The table below shows the result tc^5 of the first five effective D-intersections, up to but not including D-intersection with the primitive D-cubes of the block with output 10.

	1	2	3	4	5	6	7	8	9	10
B	1	D			D					
D			0		D	D				
E						D	$\bar{D}$			
I	1					D		$\bar{D}$		
$\cap K$				0			D		D	
tc^5	1	D	0	0	D	D	$\bar{D}$	$\bar{D}$	$\bar{D}$	

The vector a of active coordinates consists of lines 8 and 9, both of which are inputs to block 10. This fact indicates a need for a "double" D-cube, and this is generated by the following procedure.

Cube ${}^8 1\ {}^9 1\ {}^{10} 0$ has its input coordinates 8 and 9 changed and also its output coordinate to yield the cube ${}^8 0\ {}^9 0\ {}^{10} 1$. This cube is tested to ascertain whether or not it lies in the singular cover: it does, because cube q is ${}^8 0\ {}^9\ {}^{10} 1$ or

$^{8}0\ ^{9}x\ ^{10}1$, whose intersection with $^{8}0\ ^{9}0\ ^{10}1$ is indeed $^{8}0\ ^{9}0\ ^{10}1$. This then defines the double D-cube $N = {}^{8}\mathrm{D}\ {}^{9}\mathrm{D}\ {}^{10}\bar{\mathrm{D}}$.

We now return to the subroutine for generating the test cube $tc^6 = tc^5 \cap N$,

	1	2	3	4	5	6	7	8	9	10
tc^5	1	D	0	0	D	D	$\bar{\mathrm{D}}$	$\bar{\mathrm{D}}$	$\bar{\mathrm{D}}$	
N							D	D	$\bar{\mathrm{D}}$	
$tc^5 \cap N$	1	D	0	0	D	D	$\bar{\mathrm{D}}$	$\bar{\mathrm{D}}$	$\bar{\mathrm{D}}$	D

Now the vector $\boldsymbol{a}$ of active coordinates consists solely of line 10, which is a primary output $\boldsymbol{po}$, $\boldsymbol{a} \subset \boldsymbol{po}$.

The next step is the CONSISTENCY operation. In this case, the set $\boldsymbol{g}$ of coordinates of tc^5 which have the values 1 or 0 consist solely of primary inputs $\boldsymbol{pi}$, $\boldsymbol{g} \subset \boldsymbol{pi}$, so that the CONSISTENCY operation need not be performed. This completes the computation for the test cube tc^6. This cube determines the tests $^{1}1\ ^{2}1\ ^{3}0\ ^{4}0$ for line 2 stuck-at-0, and $^{1}1\ ^{2}0\ ^{3}0\ ^{4}0$ for line 2 stuck-at-1.

A method has been developed using D-notation for identifying all failures that a test detects; that method will be described in a subsequent paper.

4. Programming the D-algorithm

A program is given here for the central part of the D-algorithm. It is written, as Fig. 5 [the foldout] reveals, using a quite restricted subset of the Iverson notation. Sections 1.2 through 1.5 of Reference 9 will be found to cover all that is used here. An alternative reference is Falkoff, Iverson, and Sussenguth, Ref. 10, pp. 198–202. Further, the present section also gives a step-by-step description of the program itself.

In contradistinction to the manual procedure of Section 3, the program here is restricted to logic circuits composed of AND's, OR's, NAND's and NOR's (the transistor-type of logic used almost universally in today's technology). This restriction would be easy to remove, however, involving as it does only a few instructions. It is also assumed that the logic blocks or lines of the circuits are labelled with integers in such a way that the number assigned to any block exceeds that of all the lines that feed it; a very simple method is sufficient for such an assignment.

In Table 1 we provide a list of the symbols used in the program, together with their meaning, given in the order of their appearance in the discussion below.

There are two parts to the algorithm. In the first, a primitive D-cube $\boldsymbol{tc}$ of the failure is recursively intersected with primitive D-cubes of logic blocks seeking to form a D-cube $\boldsymbol{tc}$ which provides a "connected chain" of D-coordinates to some primary output $\boldsymbol{po}$ (Steps 1 through 38). The second part, the CONSISTENCY operation (Steps 39 to 58), consists of intersecting this D-cube $\boldsymbol{tc}$ with the singular cover S. This amounts to "driving backward"; i.e., amounts to successively assigning values to the uncommitted coordinates (those equal to X) in a way consistent with the singular cover, if this is possible.

Step 1 is to set initial parameters m and h_m to 0. Step 2 is to load $\boldsymbol{tc}$, the test cube, with a primitive D-cube of a given failure. Step 3 defines the "activity" vector $\boldsymbol{a}$, consisting of all coordinates j whose value is equal to D or $\bar{\mathrm{D}}$; the values of $\boldsymbol{a}$ are integers, belonging to the set $\boldsymbol{\iota}$ of all integers. Referring to Step 4, $\mathfrak{d}^k$ consists of the set of logic blocks driven by block k, $\nu\mathfrak{d}^k$ means the *number* or cardinality of this set, and $\nu\mathfrak{d}^a$ is to denote the vector whose k^{th} component is this number. To complete the initialization of $\boldsymbol{fa}$, the quantity $\epsilon_a^{\boldsymbol{po}}$ is added to $\nu\mathfrak{d}^a$ in order to record the fact that any particular a_j is a $\boldsymbol{po}$.

Step 4 unconditionally branches to Step 9 (we shall return to Steps 5 through 8). Now j is the index on the coordinate number of the activity vector $\boldsymbol{a}$; it runs, therefore, from 0 to $\nu\boldsymbol{a}$, the dimension of $\boldsymbol{a}$. Thus Step 9 sets j to 0. Step 10 compares j with $\nu\boldsymbol{a}$; if $j \neq \nu\boldsymbol{a}$, then Step 11 follows, which increments j by 1, $j \leftarrow j + 1$. (On the first time in reaching Step 10, j would be 0 and thus in general less than $\nu\boldsymbol{a}$. Thus we proceed first with the chain of instructions following the branch $j \neq \nu\boldsymbol{a}$.)

In Step 12, k is the index on the set (or vector) $\mathfrak{d}^{a_j}$ of logic blocks driven by a_j. Thus k ranges from 0 to the number $\nu\mathfrak{d}^{a_j}$ of elements in $\mathfrak{d}^{a_j}$. Step 12 sets k to the initial value of 0 and Step 12.1 defines the final value of kf.

Step 13 compares the magnitude of k with $\nu\mathfrak{d}^{a_j}$. If in Step 13 $k = kf$, then we return to Step 10 where we try the next coordinate of $\boldsymbol{a}$ if possible.

If $k \neq kf$, k is incremented by 1 in Step 14. (Thus, for the first time through, $k = 1$.)

Now, to simplify notation, in Step 15 and subsequent steps we let $\mathfrak{d}_k^{a_j}$, the k^{th} successor of a_j, be denoted by s (s for successor). For the same reason in Step 16 and subsequent ones, τ^s, the set of predecessors of line s, is denoted by $\boldsymbol{p}$.

Now the algorithm as described in this program is restricted for convenience of expression to the logic blocks AND, OR, NAND, and NOR. In Step 17, the expression within oblique brackets /;;/ defines a vector $\boldsymbol{t}$. The expression ϵ_p^a is a vector obtained from $\boldsymbol{p}$ by replacing each component by a 1 or 0 according to whether it is or is not in $\mathbf{a}$. When it is, the value /;;/ is D, the right term, so that the corresponding coordinate of t would also be D. When it is not, the value of the corresponding coordinate of $\boldsymbol{t}$ is the *value* of the proposition ($\lambda_s \subseteq$ AND, NAND): 1 if logic block $\boldsymbol{s}$ is an AND or NAND, and 0 otherwise, in which case it is either an OR or NOR. Thus Step 17 determines all the values of the input coordinates of s.

Step 18 determines its output coordinate by the expression in the oblique brackets and catenates this with $\boldsymbol{t}$, to form $\boldsymbol{pdc}$, the appropriate primitive D-cube of s. Within

/;;/ now, $\lambda_s \leq$ NOR, NAND is a proposition equal to 1 or 0 depending upon whether or not the logic block of s is or is not a NOR or NAND. If it is, then the output coordinate shall be $\bar{D}$, the right member of the expression. If not, then D. Thus is ***pdc*** the primitive D-cube of s defined.

Step 19 forms the D-intersection of ***tc*** with this primitive D-cube ***pdc***. However, only the subset of the coordinates of ***tc*** corresponding to the block s are needed and these are specified by the subscript p, s. The D-intersection is called ***w***.

Now in contrast to the definition in Section 2, this ***w*** is meant to be the coordinate D-intersection and in Step 20 this is tested. Step 20 evaluates the proposition: Are any coordinates of ***w*** equal to ϕ, ψ or do both λ and μ appear? If this proposition is true (equals 1) then no D-intersection is formed and we branch to Step 28 to update ***a***, ***fa***, and j.

Step 21 tests to ascertain whether or not a μ occurs in the coordinate D-intersection ***w***. If μ does *not* occur, then, according to the rules for D-intersection, the D's and $\bar{D}$'s of ***pdc*** must be interchanged. This is accomplished in steps 22 and 23.

Step 22 defines the intermediate primitive D-cube ***ipdc***: where ***pdc*** has coordinate D, ***ipdc*** has coordinate $\bar{D}$; where ***pdc*** does *not* have coordinate D, ***ipdc*** coincides with ***pdc***. In step 23, where ***pdc*** had coordinate $\bar{D}$, respecified ***pdc*** has D; otherwise the "new" ***pdc*** coincides with ***ipdc***. If μ *does* occur (in step 21) then we skip steps 22 and 23. In either case we arrive at step 24.

Steps 24 through 27 are concerned with what quantities, if any, should be stored in case the branching process returns to this stage in the execution of the algorithm.

Steps 28 through 31 are bookkeeping operations associated with ***a***. Now ***a*** consists of those coordinates of ***tc*** which are "active", that is, coordinates which lie on the frontiers of ***tc***. The respecification of ***a*** (and ***fa***) is accomplished by use of the survivor vector ***sur***. Imagine that for each coordinate of ***a*** that is a predecessor p of s, the fanout vector ***fa*** is decremented by 1, i.e., $nfa \leftarrow (fa - \epsilon_a^p)$. Only those coordinates of "new" ***fa*** which are not zero should survive since only they have a chance of propagating. This ***sur*** is defined in step 28 as a logical vector whose i^{th} coordinate is 1 if and only if nfa_i is 1; otherwise 0. In step 29 all coordinates of ***nfa*** that are 0 are deleted, i.e., ***sur*** compresses ***nfa***, this respecifies ***a***. In step 30 ***a*** is similarly compressed by ***sur***, to eliminate all coordinates of ***a*** which are not on the frontiers of ***tc***. Step 31 updates the index j on ***a*** to compensate for the compression. For each coordinate $r \leq j$ whose corresponding entry in ***sur*** is 0, j is decremented by 1.

Step 32 decides (again) whether or not a valid D-intersection was formed. If it was, then we branch to step 35 to form the new ***tc***. If it was not, step 33 tests to see whether or not the D-intersection failed because of a D and $\bar{D}$ both being present in tc_p. If a D-, $\bar{D}$-combination does not occur, we return to step 13 to resume the examination of successors. When both a D and $\bar{D}$ are present, step 33.1 tests to see if the block output has previously been set to the incorrect value: if so, it is necessary to back up, thus a branch to step 6 is executed. Step 33.2 tests to see if the output is X. If it is, step 34 is used to set the output. If the output is not an X, it must already have been set correctly; the program branches to step 13.

Now what we want of new ***tc*** is that its old coordinates shall be those of old ***tc*** while its new shall be those associated with block s, namely either its input coordinate p or its output s. Thus the expression within the left oblique bracket \;;\ (which Iverson calls *mesh*) contains a vector $\epsilon^{p,s}$. If the r^{th} coordinate of this vector (or proposition) is 1, then the corresponding coordinate of ***tc*** is that of ***w***. If it is 0, then the corresponding coordinate of ***tc*** is that of old ***tc***, which is obtained by restricting ***tc*** to the subset $\bar{\epsilon}^{p,s}$, *not* in the inputs or outputs of the block s. This is what $\bar{\epsilon}^{p,s}/tc$ specifies.

Step 36 adjoins s to ***a***. In step 37, necessary modifications for ***fa*** are made (Cf. step 3).

Step 38 determines whether or not the "D-drive" to the output has been completed, i.e., whether all entries in ***a*** are primary outputs which have been driven forward as far as possible. If not, the action returns to step 13, to resume the examination of successors. If it has, then we have pursued the D's to primary outputs and we have finished with the first part of the algorithm.

The only instructions in the first part of the algorithm yet to be explained are steps 5 through 8. These are reached through the backup procedure in branching. Step 5 is accessed from step 10 when all elements in ***a*** have been examined. Step 5 ascertains whether or not a ***po*** has been reached. If it has, we branch to step 38. If a ***po*** has not been reached, we ask in step 6 whether any branching levels remain to be investigated: i.e., is $m = 0$? If not, then no test exists. If so, steps 7 and 8 back up in the branching structure.

The last part of the algorithm, the CONSISTENCY operation, begins with step 39. This step defines a vector ***g*** consisting of all lines whose coordinates of ***tc*** have values 1 or 0. Step 40 asks whether or not ***g*** consists solely of primary inputs ***pi***. If it does, the algorithm STOPS, for the purpose of the CONSISTENCY operation is consistently to drive back the values on the lines ***g*** to primary inputs, to determine the actual test, in terms of primary inputs, to the circuit. Step 41 deletes those coordinates of ***g*** which are already primary inputs. Since we will always be working with the last element of ***g***, step 42 sets the ***g*** index, n, to vg. Step 43 tests the inputs (predecessors) of logic block g_n to see if both a D and $\bar{D}$ occurs in ***tc***. If both do occur, this block can be skipped (its output was determined strictly by the D, $\bar{D}$; cf. step 34) so that step 44 deletes g_n from ***g*** and branches back to step 40 where the completion test is

made. If they do not both occur, step 45 is entered. Step 45 sets to 0 the index l on the number of rows of the singular cover ${}^{r}S$ of the r^{th} logical block. As was indicated in Section 3, it is necessary that S consist of prime cubes, that is, of prime implicants. In Step 46 $\mu({}^{g_n}S)$ is the number of rows of the matrix ${}^{g_n}S$ and Step 46 tests whether or not all rows of ${}^{g_n}S$ have been tried (unsuccessfully). If they have, then this test cube tc is finished and control shifts to Step 55 where m is tested to ascertain whether all branches have been exhausted. If $m = 0$, the process is ended and there is no test. If $m \neq 0$, then in Step 56 the parameter h_m is tested. If $h_m = 1$ then the next branch is in the CONSISTENCY mode and Steps 57 and 58 prepare for a return to Step 46. But if $h_m = 0$ in Step 56 control shifts to Step 7 for commencing the first part of the algorithm. But we return to Step 46. If l is less than $\mu({}^{g_n}S)$ then more D-intersections can be formed: Step 47 increments l and Step 48 forms the coordinate D-intersection. Step 49 is a test to ascertain whether or not a nontrivial D-intersection is formable. If not, control returns to Step 46. If so, then Steps 50, 51, 52 ascertain whether to save this stage of the computation: if $l < \mu({}^{g_n}S)$ then it is saved. In any event in step 53, g_n is deleted from g and those coordinates of ${}^{g_n}S$, whose value is 0 or 1 and for which $tc = x$, are added to g. Step 54 respecifies tc by y. Finally control is shifted to Step 40 where the test for completion is made: is g contained in the set pi of primary inputs? With this, then, the description of the program is complete.

5. Proof of validity of algorithm

•. *Definition*: Let $\eta \equiv \eta^1, \eta^2, \cdots, \eta^r$ be a set of D-cubes. Let $\partial(\eta)$ be defined as their D-intersection

$$\partial(\eta) = \eta^1 \cap \eta^2 \cap \cdots \cap \eta^r.$$

If η is the empty set ϕ, let $\partial(\eta) = \text{xx} \cdots \text{x}$, of dimension vG.

For notational convenience let $c(T, F)$ be denoted c. As in Section 4 we restrict ourselves to the functions AND, OR, NAND, and NOR for the logic blocks λ.

•. A-*construction*: Let I be a logic block not the site of the failure F, for which the output coordinate c_I of c is D or $\bar{\text{D}}$. We define a D-cube α^I in the following manner: for the immediate inputs p of I and output I, let the coordinates of α^I coincide with those of c; let the coordinates of α^I elsewhere be x. The proof of Lemma 6 establishes that α^I is a primitive D-cube of block I.

Let α denote the set $\alpha^{I_1}, \cdots, \alpha^{I_m}$ of all such primitive D-cubes, one for each block I, which has a D or $\bar{\text{D}}$ in its output coordinate and is not the site of failure F.

•. B-*construction*: Suppose that for logic block J the output coordinate of c is 1 (or 0) and both D and $\bar{\text{D}}$ do not occur on the inputs. Then c can be used to form a singular cube β^{J*} according to the following construction. Let r be an output or input coordinate of block J: if $c(r) \equiv$ 1 (or 0), let $\beta^{J*}(r) \equiv 1$ (or 0); if $c(r) =$ D, $\bar{\text{D}}$ or x, let $\beta^{J*}(r) =$ x; let all other coordinates of β^{J*} be x.

Lemma 7: (1) The cube β^{J*} is contained in a singular cube β^J of ${}^{J}S$. (2) Furthermore, β^J D-contains $c(T, F)$.

Proof: We shall prove part (1) of the lemma for λ_J being an AND. Similar proofs establish the lemma for the other three possibilities.

Case I: The output of block J is a 1. This implies that all inputs are 1 and the lemma immediately follows.

Case II: The output of block J is a 0. Now the only ways in which it is possible in the D-algorithm for this output to be 0 is that either (a) a D-, $\bar{\text{D}}$-combination occurs, which is ruled out by hypothesis or (b) some input z of J is 0. Clearly β^{J*} is contained in the prime cube β^J of ${}^{J}S$, consisting of a 0 in lines z and J, and an x in all other coordinats.

Part (2) of the lemma follows immediately from the construction. *Q.E.D.*

Let β denote the set $\beta^{J_1}, \cdots, \beta^{J_n}$ of such constructed singular prime cubes, one for each block J which has in c a 1 or 0 in its output and whose input does not contain both D and $\bar{\text{D}}$.

•. Γ-*construction*: Suppose that for logic block K its output coordinate in $c(T, F)$ is 1 (or 0) and that a D and $\bar{\text{D}}$ occur as input coordinates. In this case we define cube γ^K consisting of a 1 (or 0) in coordinate K and x's elsewhere. Let $\gamma = \gamma^{K_1}, \cdots, \gamma^{K_q}$ be the set of all such γ^{K_i}s.

First we state

Hypothesis H: Given a Boolean graph G, a failure F in G, and the existence of a test T to detect F in G.

Theorem 1: Under Hypothesis H, the following conclusions hold:

(1) The D-cube of test-and-failure $c(T, F)$ (as defined in Section 2) can be constructed.

(2) This D-cube $c(T, F)$ conversely defines T as its primary input coordinates pi.

(3) There exists a primitive D-cube-of-failure α^0 which D-contains $c(T, F)$

$$\alpha^0 \supset c(T, F).$$

(4.0) There exists a set $\alpha \equiv \alpha^{I_1}, \alpha^{I_2}, \cdots, \alpha^{I_m}$ of primitive D-cubes of a set of logic blocks in G such that their D-intersection $\partial(\alpha)$ D-contains $c(T, F)$,

$$\partial(\alpha) \supset c(T, F).$$

(4.1) There exists a set β of prime singular cubes $\beta^{J_1}, \beta^{J_2}, \cdots, \beta^{J_n}$ of G such that $\partial(\beta)$ D-contains $c(T, F)$,

$$\partial(\beta) \supset c(T, F).$$

(5) Let $c^* \equiv \alpha^0 \cap \partial(\alpha) \cap \partial(\beta) \cap \partial(\gamma)$. Then c^* D-contains $c(T, F)$,

$$c^* \supset c(T, F),$$

and c^* defines a set T^* of tests containing T.

Proof of Theorem 1

(1) The construction of $c(T, F)$ requires that, for each line of G in both the good and failing circuit, a unique signal value occurs. This is ensured by the hypothesis that G is a Boolean graph.

(2) By definition, T is completely specified by the values of the coordinate positions of the primary inputs of $c(T, F)$.

(3) This is a restatement of Lemma 4, with α^0 being the primitive-D-cube-of-failure.

(4.0) By Lemma 6 for each logic block I, for which an output coordinate of $c(T, F)$ is D or $\bar{D}$, there is a primitive D-cube of failure $\alpha^I \supset c(T, F)$. That $\partial(\alpha) \supset c(T, F)$ follows directly from Lemmas 3 and 5.

(4.1) By Lemma 7 for each logic block J for which the output coordinate of $c(T, F)$ is a 1 or 0 and for which a D–$\bar{D}$-combination does not appear on its inputs, a singular cube β^J is constructed, each D-containing $c(T, F)$. Let there be n of these blocks J, and thus n β's: $\beta^{J_1}, \cdots, \beta^{J_n}$. By Lemmas 3 and 5, $\partial(\beta) = \beta^1 \cap \beta^2 \cap \cdots \cap \beta^n \supset c(T, F)$.

(5) That $c^* \supset c(T, F)$ follows directly from the A-, B- and Γ-constructions and from Lemma 5. Now since $c^* \supset c(T, F) \equiv c$, the only way in which c^* differs, if at all, from c in *any* coordinate r is for c^*_r to be x while c_r is 1 or 0. Thus, denoting c^*_{pi} as T^* and c_{pi} as T, we have that $T^* \supset T$. *Q.E.D.*

Theorem 2: If for a given failure F of the circuit there exists a test T to detect that failure, then the D-algorithm will compute a test cube $c(T', F)$ for some test T'.

Proof of Theorem 2.

Assume that, in the execution of the D-algorithm, as described in Fig. 5, no test cube $tc \equiv c(T', F)$ has been constructed before the test cube $c(T^*, F)$ has been encountered, for if one such has been, then the theorem is already proven. We shall demonstrate, under this assumption, that $c(T^*, F)$ is in fact generated by the D-algorithm.

Assume that the set $\eta = \alpha \cup \gamma$ has been arranged in ascending order with respect to superscripts, logic block numbers, i.e., $\eta = \eta^{L_1}, \cdots, \eta^{L_{m+q}}$ where $L_i < L_{i+1}$. Let β be similarly ordered.

The following terms will be convenient in the exposition. Block w is said to be a D-*successor* of block v if and only if block w has a D or $\bar{D}$ in the coordinate position corresponding to block v. We shall dually describe this relationship by saying that block v is a D-*predecessor* of block w.

A sequence of sets H^v (composed of elements from α^0, η, and β) and corresponding test cubes tc^v, $v = 0, 1, \cdots$, $n + m + q$, will now be constructed, recursively, in such a manner that the tc^v will correspond directly to a sequence of test cubes generated by the algorithm in Fig. 5.

Let $H^0 = \alpha^0$ and $tc^0 = \partial(H^0) \equiv \alpha^0$. Having defined the set H^v and test cube tc^v we proceed to define H^{v+1} and tc^{v+1} from η^v, the v^{th} entry in H^v, as follows.

Case 1: Suppose η^v is a γ. Then

$$H^{v+1} \equiv H^v$$

$$tc^{v+1} \equiv tc^v;$$

i.e., in the algorithm the successors of blocks arising from γ's are never examined.

Case 2: Suppose η^v is an α. Then

$$H^{v+1} = H^v \cup \Sigma(\eta^v)$$

$$tc^{v+1} = tc^v \cap \partial[\Sigma(\eta^v)],$$

where $\eta^w \in \Sigma(\eta^v)$ if and only if: (a) η^w is not an element of H^v; (b) η^w is a D-successor of η^v; and (c) one of the following two conditions holds:

Condition 1. Suppose η^w is an α. Then all D-predecessors of w must have their corresponding primitive D-cubes of failure, α, in H^v.

Condition 2. Suppose η^w is a γ. Then there must be at least one D-predecessor of block w whose primitive D-cube of failure is in H^v and has value *opposite* to that of η^v in its output coordinate.

After $v^* = m + q$ iterations, H^v contains all the elements of α_0, α, and γ. This corresponds to the completion of the D-drive: the η^v do not have any new D-successors, and the primary outputs *po* must therefore have been reached.

The final n iterations in the formation of H^v and tc^v, where $v^* < v \leq v^* + n$, are carried out as follows. Pick the highest coordinate position, hence a logic block, in tc^v which is a 1 or 0, which is not a primary input *pi*, and which has not already been examined (Cf. Steps 41 and 42 of Fig. 5). If there is both a D and a $\bar{D}$ on the inputs of this block skip the coordinate. If not, then the B-construction generated a β^w for this block. Now form

$$H^{v+1} = H^v \cup \beta^w$$

$$tc^{v+1} = tc^v \cap \beta^w.$$

After n iterations, this final intersection process must terminate.

We can now make the following assertions:

1. Theorem 1 proves that $c^* \equiv \alpha^0 \cap \partial(\alpha) \cap \partial(\beta) \cap \partial(\gamma)$ defines a set of tests T^* containing T.
2. The above construction follows precisely the steps of

the D-algorithm of Fig. 5 in generating the test cube tc^{m+n+q}: the order of examination of η^v and formation of $\sum(\eta^v)$ corresponds precisely to that specified by Steps 5 through 38 and the order of intersection of the β's is specified by Steps 39 through 58.

Since intersection is commutative (Lemma 3) we have

$$tc^{n+m+q} = c^*.$$

Thus the algorithm generates a test T^* which contains T.

Q.E.D.

Acknowledgement

The author is most grateful for the manifold contributions of P. R. Schneider to the final formulation of this paper; specifically, the structure and proof of Theorems 1 and 2 are the joint work of JPR and PRS. Further, the final version of the programmed algorithm (Figure 5 and Section 4) was a joint undertaking of JPR, PRS, and W. G. Bouricius. The author is much indebted to E. G. Wagner who read critically an earlier version of the paper and who continued to make penetrating observations. The author is also endebted to K. E. Iverson and Adin Falkoff who, at a still earlier stage, helped him in writing his first version of the Iverson-notation program. Finally, thanks are due to B. O. Evans, then Vice President of the IBM Data Systems Division, who first urged the preparation of such a paper for the IBM Journal.

References

1a. W. C. Carter, H. C. Montgomery, R. J. Preiss, and H. J. Reinheimer, "Design of Serviceability Features for the IBM System/360," *IBM Journal* **8**, 2 (1964).

1b. K. Maling and E. L. Allen, Jr., "A Computer Organization and Programming System for Automated Maintenance," *IEEE Trans. Elect. Comp.* **EC-12**, No. 6, 887–895 (December, 1963).

2. J. P. Roth and R. M. Karp, *Minimization over Boolean Graphs, IBM Journal* **6**, 2 (1962).

3. R. E. Miller, *Switching Theory, Vol. I*, John Wiley and Sons, New York, 1965. p. 233.

4. J. P. Roth, "Algebraic Topological Methods in Synthesis," in *Proceedings of an International Symposium on the Theory of Switching, 2–5 April* 1957, *Part I:* The Annals of the Computation Laboratory of Harvard University, Vol. XXIX, pp. 57–73, Harvard University Press, Cambridge, Mass., 1959.

5. J. M. Galey, R. E. Norby, and J. P. Roth, "Techniques for the Diagnosis of Switching Circuit Failures," *Transactions of the IEEE Communications and Electronics* **83**, 74, Sept. 1964, pp. 509–514.

6. C. B Stieglitz, Paper in preparation.

7. J. P. Roth, "A Pragmatic Theory of Automata," Lecture given at *A Symposium on Switching Theory and Automata*, International Federation of Automatic Control, Moscow, 1962. Cf. also IBM Data Systems Division Report TR00.918, Sept. 21, 1962, Poughkeepsie, N. Y.

8. J. P. Roth, "Algebraic Topological Methods for the Synthesis of Switching Systems, I.", *Transactions of the American Mathematical Society* **88**, 2, 301–326, July 1958. Cf. also Institute for Advanced Study, Princeton, N. J., ECP 56–02, April 1956 and General Electric Company Report No. R55GL345, Schenectady, N. Y., Sept. 1955.

9. K. E. Iverson, *A Programming Language*, John Wiley & Sons, New York, 1963.

10. A. D. Falkoff, K. E. Iverson and E. H. Sussenguth, "A Formal Description of System 360," *IBM Systems Journal* **3**, 3, 1964, pp. 198–263.

Received October 15, 1964

Revised Manuscript received March 14, 1966.