A realistic approach to detection test set generation for combinational logic circuits

R. G. Bennetts

Department of Electronics, University of Southampton, Southampton SO9 5NH

The paper presents a technique for deriving a detection test set for combinational logic circuits, that is based on the exclusive-OR operator. It is claimed that the procedure is suitable for implementation on a digital computer and this is supported by results relating to single-/multi-output circuits using either basic gates or MSI/LSI logic elements.
(Received January 1972)

Before a combinational logic circuit can be functionally tested, a suitable set of tests must be derived. Other than exhaustive enumeration, the derivation of each test in this set may be based on postulating the existence of a single stuck-at fault and determining an input stimulus, the response to which differs under the fault/no fault condition. By successively postulating all such single logical faults on all connections, primary and internal, a full set of tests may be evaluated.

Obviously, for an n-input circuit, application of all 2^n input combinations, accompanied by a comparison with, say, a truth table or master copy, will certainly provide complete checkout and this approach is used by a number of logic testing systems. The disadvantage becomes apparent as the number of primary inputs n increases. For instance, assuming a testing system with a stimulus/response cycle of 1 μs, it would take 18 hours to exhaustively test a 30 primary input logic circuit, assuming no fault.

Fortunately, such a test set would contain a large amount of redundancy, since in practice, a test will provide cover for more than one fault condition. Consequently, a suitably selected subset can be specified that affords the same degree of detection resolution. This in turn requires a selection process and the fault matrix technique was developed to achieve this. (See Bennetts and Lewin (1971) and Bennetts (1971) for more detailed discussion of the fault matrix and subsequent techniques mentioned, but not expanded, in this paper.) The matrix contains information relating the 2^n inputs with the outputs expected for both the no-fault and faulty conditions. Row and column dominance techniques are then used to select a test subset such that all faults are covered at least once.

An advantage of this technique is that the so-called cover problem has received a lot of attention, principally in connection with the minimisation of combinational logic functions (Quine-McCluskey algorithm, etc.) and current algorithms are both efficient and speedy. One disadvantage, however, is that the entries in the matrix have to be determined and it is usual to employ a logic simulator.

Another disadvantage, from a practical point of view, is the sheer size of the matrix as n becomes large (>30 say). In terms of computer core store, a matrix containing 2^{30} rows ($\sim 10^9$) and $(2S + 1)$ columns, for a total of S primary inputs, internal connections and primary outputs, is quite massive and any subsequent manipulation is going to take some time. As a result therefore, a different strategy must be sought and, for combinational circuits, the two major techniques available are those referred to as n-dimensional path sensitisation (D-algorithm) and Boolean Difference (Bennetts and Lewin, 1971; Bennetts, 1971). Both these techniques offer significant advantages over the fault matrix technique and this paper presents a procedure that has been developed and programmed, based mainly on the Boolean difference technique, but modified by path sensitising concepts. One of the major considerations in forming this procedure was that it should be realistic in the sense that a computer program version should be able to handle fairly large circuits, i.e., 55 primary inputs, 7 primary outputs and 100 logic gates. In practice, a circuit this size has been analysed and a suitable detection test set derived.

Theoretical considerations of the algorithm

The basis of the algorithm is the use of the exclusive-OR operator on the Boolean function representing the logic circuit (Sellers, Hsiao, and Bearnson, 1968a). Briefly, if an n-input circuit is described by:

$$z = f(x_1, x_2, \ldots, x_i, \ldots, x_n) \quad (1)$$

then a new function representing the logical behaviour under the x_i faulty condition may be similarly defined:

$$z_{x_i} = g(x_1, x_2, \ldots, \bar{x}_i, \ldots, x_n) \quad (2)$$

in which $x_i(\bar{x}_i)$ in z is replaced by $\bar{x}_i(x_i)$ in z_{x_i}.

The Boolean difference, written $Dz(x_i)$ is then defined:

$$\begin{aligned} Dz(x_i) &= z \oplus z_{x_i} \\ &= f(x_1, x_2, \ldots, x_i, \ldots, x_n) \oplus g(x_1, x_2, \ldots, \bar{x}_i, \ldots, x_n) \\ &= h(x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \end{aligned} \quad (3)$$

Note one important fact here—namely that $Dz(x_i)$ *is independent of* x_i—(see Appendix 1 for a formal proof of this).

Note also that if $Dz(x_i) = 0$, then z is independent of x_i, i.e., x_i is logically redundant.

In order to actually derive the test conditions for x_i faulty, we need to identify those tests that relate to x_i stuck-at-1, (s-a-1) and x_i stuck-at-0 (s-a-0). Since $Dz(x_i)$ is independent of x_i, ANDing $Dz(x_i)$ with x_i will create a set of tests, all of which will demand a 1 on x_i and therefore determine test conditions for x_i s-a-0. Similarly, ANDing $Dz(x_i)$ with $\bar{x}_i$ will create the s-a-1 tests, i.e.,

$$Dz(x_i).x_i = \text{s-a-0 test set} \quad (4)$$

$$Dz(x_i).\bar{x}_i = \text{s-a-1 test set} \quad (5)$$

In general, provided $Dz(x_i) \neq 0$, it may be expressed in a sum-of-products (s-o-p) form. We will now consider the significance of each term in such an expression.

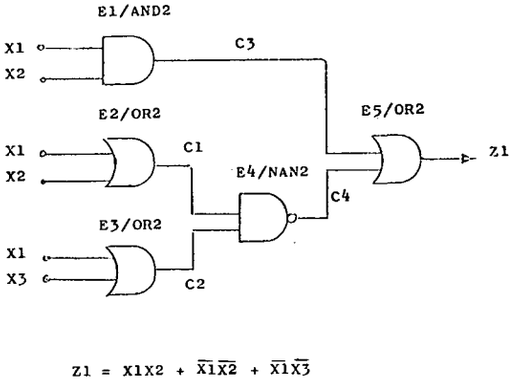

Fig. 1. Example circuit

Fault		Tests
X1	s-a-0	$X1\ \overline{X2} + \overline{X1}\ X3$
X1	s-a-1	$\overline{X1}\ \overline{X2} + \overline{X1}\ X3$
X2	s-a-0	$X1\ X2 + X2\ X3$
X2	s-a-1	$X1\ \overline{X2} + \overline{X2}\ X3$
X3	s-a-0	$\overline{X1}\ X2\ X3$
X3	s-a-1	$\overline{X1}\ X2\ \overline{X3}$

Fig. 2. Tests for $X1$, $X2$ and $X3$

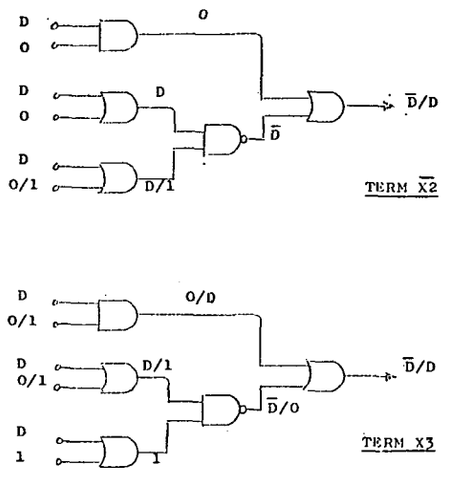

Fig. 3. Sensitive paths established by $DZ1(X1)$

Consider the circuit shown in **Fig. 1.** For simplicity, the following convention is used throughout this paper.

Element numbers are preceded by	E
Primary input lines	X
Primary output lines	Z
Internal lines	C

Element types are up to four alphanumeric characters.

By Karnaugh mapping or otherwise, the following expressions are true:

$$DZ1(X1) = \overline{X2} + X3 \tag{6}$$

$$DZ1(X2) = X1 + X3 \tag{7}$$

$$DZ1(X3) = \overline{X1}X2 \tag{8}$$

From equations (4) and (5) therefore, the following tests may be defined—see **Fig. 2.** If we consider the s-o-p expressions above (equations (6), (7) and (8)), we note that they are in their minimal normal form. In some cases this has resulted in 'don't care' literals in the tests specified in Fig. 2. We can further recognise that each term in the Boolean differential s-o-p represents a specification that establishes a sensitive path or paths from the primary input in question to the primary output. For instance, the sensitive paths set up by the two terms in $DZ1(X1)$ are shown in **Fig. 3**, Here the D is used in the same sense as in Roth's Calculus of D-cubes (Roth, 1966), i.e., it may assume either the value 0 or 1, but whichever it is, it remains consistent throughout the circuit.

Consideration of Fig. 3 raises another factor—namely that the assignment of the don't cares can have an effect on which sensitive paths are actually selected. In connection with this, we are primarily interested in the assignment that results in the maximum internal fault coverage*, and we note that a test will cover not only the fault on the primary input line to which it relates, but also any fault on the sensitive path, including the primary output. This does not necessarily mean however that assignments that establish multiple paths, i.e., paths that originate from a single source and eventually reconverge again, will have an increased fault coverage, since those faults along each multiple section of the path may not in fact be covered. This is due to the fact that the reconverging action may rely on the effect of the fault propagating through all paths simultaneously. This is demonstrated in Fig. 3 for the input case $D00$ on $X1X2X3$ respectively. Here $X3$ has been assigned 0 since it appears to sensitise two parallel paths (through $E2$ and $E3$, reconverging at $E4$). The fault coverage of these two tests ($D = 0$, $D = 1$) is:

$\overline{X1}\ \overline{X2}\ \overline{X3}$ covers $X1$ s-a-1, $C4$ s-a-0, $Z1$ s-a-0
$X1\ \overline{X2}\ \overline{X3}$ covers $X1$ s-a-0, $X2$ s-a-1, $C1$ s-a-0
$C2$ s-a-0, $C3$ s-a-1, $C4$ s-a-1
and $Z1$ s-a-1

The increased coverage of the second test $X1\ \overline{X2}\ \overline{X3}$ is due to two facts:

1. It is also a test for $X2$ s-a-1 and sets up a sensitive path through $E1$ and $E5$ thus covering $C3$ s-a-1 and $Z1$ s-a-1.
2. Despite the parallel paths through $E2$ and $E3$, the reconverging effect on $E4$ is such as to cause all its inputs to be at a control status†. Consequently, any single change in any of $E4$'s inputs resulting from a fault on one of the parallel paths, has an observable effect on the reconverging gate, and is thus covered.

   In the $\overline{X1}\ \overline{X2}\ \overline{X3}$ case, all inputs to $E4$ were static and not until they all change, is an effect observable on $E4$ output. This means that any single input change can have no effect and limits fault coverage to those sections of the sensitive paths that are not divergent—in this case, $X1$, $C4$ and $Z1$ only.

Returning to our consideration of the terms therefore, we can now make the following observations:

1. If a single term in $Dz(x_i)$ is completely specified, except for x_i, then that term will establish a unique sensitive path from x_i to z. Such a term will be referred to as 'completely specified -x_i' or shortly CS-x_i.
2. If the *minimal normal form* of $Dz(x_i)$ is such that each term is CS-x_i, then each term will specify an alternative sensitive path such that *all* possible sensitive paths emanating from x_i and terminating at z must be so specified.

Intuitively this is true since if there exists a sensitive path from x_i to z that is not specified by one of the CS-x_i terms in $Dz(x_i)$, then $Dz(x_i)$ is incomplete.

3. As a result of 2, inclusion of all such tests in the detection test set will guarantee cover on all internal faults occurring on lines that are functionally dependent on x_i. Consequently if this is carried out for all x_i, $1 \leqslant i \leqslant n$, then the final set of tests can be guaranteed to cover all internal faults on lines that are functionally dependent on the primary input set, i.e., all irredundant logic lines.
4. If the minimal normal form of $Dz(x_i)$ contains terms that are not CS-x_i, then such terms can obviously be expanded to render them CS-x_i. If this is carried out, observations 2 and 3 are still true with the modification that, in 2, each sensitive path may not be unique, i.e., paths may be repeated by other terms, and consequently, in 3, the tests may contain redundancies.

If, however, the minimal normal form is not expanded, i.e.,

*For instance, t_6 is not a valid test condition for $X1$ s-a-0, but is for $X11$ s-a-0, provided $X12$ and $X13$ are both logically correct, i.e. at logical 1.

†This is true for detection test sets, but not for detection *and* location. In the latter case, the requirement is reversed.

contains terms that are not $CS\text{-}x_i$ ($NCS\text{-}x_i$), then the assignment of don't cares has an effect on which sensitive path, and thus internal fault coverage, is established. This in turn implies that, if each $NCS\text{-}x_i$ term is used to generate two tests with, say random assignment of don't cares, then the resulting $2m$ tests (from m terms) does not guarantee to cover all faults on internal lines functionally dependent on x_i.

It is thus conceivable that if the full test set is based on minimal normal forms that contain $NCS\text{-}x_i$ terms, with random assignment of don't cares, then it may not provide full internal fault coverage.

On the other hand, a test set based on normal forms that have been expanded, if necessary, to full $CS\text{-}x_i$ status, will potentially contain a number of redundancies and may, in the limit, specify all 2^n input configurations. Consequently, such a scheme is considered impractical and the algorithm to be described is based on the generation of terms in the near-minimal normal form expressions for $Dz(x_i)$ with, currently, random assignment of don't cares. There is obviously more work to be done in the assignment criteria for don't care terms and this is intimately connected with the actual structure of the circuit.

Before describing the algorithm proper, one more observation can be made:

5. It has already been shown that the sensitive paths established by any test is fundamentally a function of the structure of the circuit. In particular, the basic fanout of each primary input is a loose lower limit on the minimum number of tests per primary input necessary to ensure exercising all possible sensitive paths starting from x_i. In other words, with reference to $X1$ in Fig. 1, there are three separate gates, $E1$, $E2$ and $E3$, to which $X1$ becomes an input. There are potentially therefore, at least three input conditions which will cause each one of the gates, in turn, to become a member of a sensitive path. In practice, of course, one of three things can happen:
   (*a*) Two or more sensitive paths may be formed in parallel resulting in at least one redundant test.
   (*b*) The three tests do in fact exercise each gate alone, in turn.
   (*c*) Internal fanout on one or more of the separate sensitive paths requires more than three tests to provide full coverage.

As a result therefore, the heuristic approach of generating p_i terms in $Dz(x_i)$ where p_i = the fanout of x_i, is rather weak. Nevertheless, it offers a compromise between the two extremes of one term per x_i to the full $CS\text{-}x_i$ set, and is the approach that is suggested in the algorithm. The primary input fanout can be handled on a more theoretical basis and this aspect is explored further in Appendix 2—q.v.

The detection test set algorithm

The description of the algorithm will be illustrated where necessary, with the circuit example in Fig. 1.

Stage 1—*Derivation of the s-o-p expression*

The starting point is with the topological description of the logic circuit. This is essentially a list of each logic element accompanied by its type, output and inputs. For example, the circuit drawn in Fig. 1, could be described as shown in **Table 1.**

In order to apply exlusive —OR techniques to such a circuit, it is necessary to derive the s-o-p form for $Z1$ as a function of its inputs.

One way of achieving this is by logic simulation to define the truth table, followed by some combinational minimisation procedure to yield a reduced s-o-p expression. This has its obvious disadvantages and an attractive alternative is to describe the output of each element in terms of its inputs, in a Polish format, and then by successive substitution, to build up a Polish expression relating $Z1$ to its primary inputs alone.* This process is eased if the order of the elements is arranged to suit. By this is meant that if Ci is both the output of Ej and the input to Ek, then Ek precedes Ej. A suitable reordering of the elements in Table 1 is:

$$E5, E4, E3, E2, E1 .$$

This has the advantage now of actually specifying which form of the output of each element is required, i.e., TRUE and/or FALSE, and thus ensures that only those Polish expressions necessary for each element are formed.

With the reordering suggested the elemental Polish expressions derived are shown in **Table 2.**

Here, the operators are restricted to AND and OR only, and the TRUE outputs of inverting gates (NAND or NOR) or FALSE output of non-inverting gates (AND or OR), are reconfigured into their s-o-p form before forming the Polish, e.g., the output of NAND gate $E4$ is given by:

$$\begin{aligned} C4 &= \overline{C1}\ \overline{C2} \\ &= \overline{C1} + \overline{C2} \\ &\rightarrow \overline{C1}\ \overline{C2}\ \text{OR(2) in Polish} \end{aligned}$$

The general form of each operator in the Polish is AND(i) or OR(i). The (i) specifies the number of literals or terms to be ANDed or ORed together. This will become more explicit later.

Table 1 Topological description of the circuit

$E1$/AND2/$C3$/$X1$, $X2$
$E2$/OR 2/$C1$/$X1$, $X2$
$E3$/OR 2/$C2$/$X1$, $X3$
$E4$/NAN2/$C4$/$C1$, $C2$
$E5$/OR 2/$Z1$/$C3$, $C4$

Table 2 Elemental Polish expressions

ELEMENT	POLISH EXPRESSION
$E5$	$Z1 = C4\ C3$ OR (2)
$E4$	$C4 = \overline{C2}\ \overline{C1}$ OR (2)
$E3$	$\overline{C2} = \overline{X1}\ \overline{X3}$ AND (2)
$E2$	$\overline{C1} = \overline{X1}\ \overline{X2}$ AND (2)
$E1$	$C3 = X1\ X2$ AND (2)

Table 3 Stages in the unpacking procedure

$\overline{X3}\ \overline{X1}$ AND (2) $\overline{X2}\ \overline{X1}$ AND (2) OR (2) $X2\ X1$ AND (2) OR (2)

RESULT IN STACK AFTER: 1ST OPERATOR	2ND OPERATOR	3RD OPERATOR
$\overline{X1}\ \overline{X3}$	$\overline{X1}\ \overline{X2}$ $\overline{X1}\ \overline{X3}$	$\overline{X1}\ \overline{X2} + \overline{X1}\ \overline{X3}$
4TH OPERATOR	5TH OPERATOR	
$X2\ X1$ $\overline{X1}\ \overline{X2} + \overline{X1}\ \overline{X3}$	$X2\ X1 + \overline{X1}\ \overline{X2} + \overline{X1}\ \overline{X3}$	

*Strictly speaking, the Polish expressions are in a *reverse* Polish form.

Based on the elemental Polish expressions, successive substitution for C-types in the $Z1$ expression produces the following overall expression:

$$Z1 = \overline{X3}\ \overline{X1}\ \text{AND(2)}\ \overline{X2}\ \overline{X1}\ \text{AND(2)}\ \text{OR(2)}\ X2\ X1\ \text{AND(2)}\ \text{OR(2)} \quad (9)$$

and in conjunction with this, we note that the fanouts of $X1$, $X2$ and $X3$ are 3, 2, and 1 respectively.

The next stage is to unpack this Polish expression into its s-o-p form, and here we require a minimal or near-minimal version. The reason for this will become clearer when we discuss generation of $Dz(x_i)$ terms but suffice it to say that the speed of the process and amount of core store required is directly dependent on the minimality or otherwise of this expression.

The unpacking procedure is essentially a last-in, first-out stack technique, in which all literals are pushed down into a stack and an operator, AND(i) say, will then operate on the last i terms in the stack. This procedure, although in essence very simple, is complicated by the need to produce the result in a s-o-p format, rather than any other format. This means for instance, that if an AND operator is called with two or more ORed expressions, the full expansion must be carried out and this in turn will require some checking procedure to remove terms absorbed by others.

For example:

$$X1\ X2\ \text{OR(2)}\ X1\ X3\ \text{OR(2)}\ \text{AND(2)}$$
$$\rightarrow (X1 + X2)(X1 + X3)$$
$$= X1 + X1\,X2 + X1\,X3 + X2\,X3$$
$$= X1 + X2\,X3 \text{ after the absorption check.}$$

The various stages in unpacking the Polish expression of equation (9) are shown in **Table 3.**

The final stage produces the required s-o-p expression and the repeated application of the absorption check will ensure minimality or near-minimality of the result*.

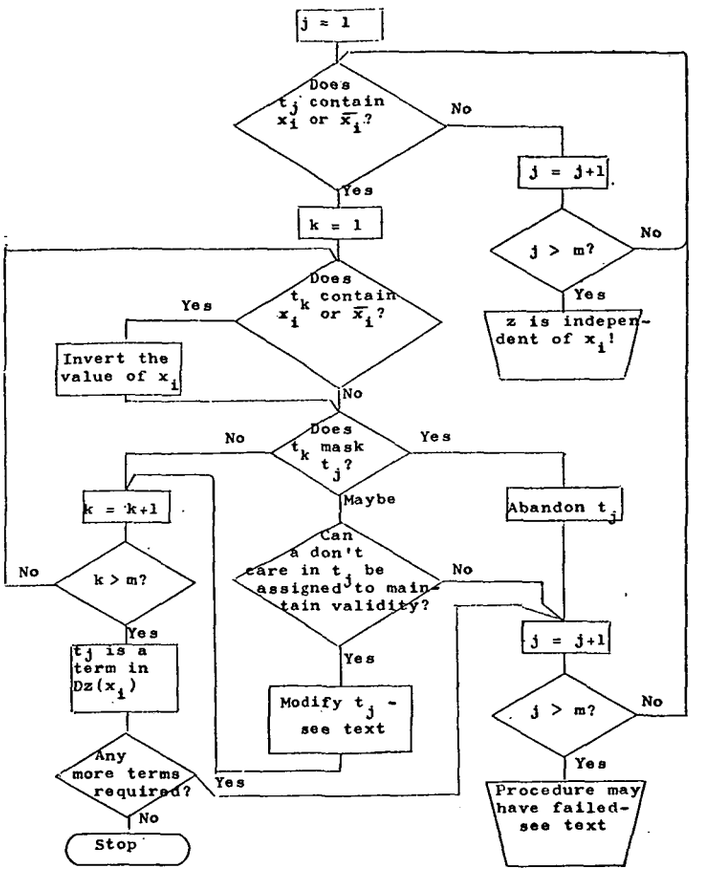

Fig. 4. Flow chart of $Dz(x_i)$ term algorithm

Stage 2—Generation of terms in $Dz(x_i)$

The next stage is to generate the required number of terms in $DZ1(X1)$, $DZ1(X2)$ and $DZ1(X3)$.

Theoretically it is possible to derive these Boolean differential expressions using, for example, the theorem quoted in Appendix 1(q.v.). This leads to obvious processing difficulties due to the inversion requirements of the s-o-p expressions defined by $f_1(X_i)$, $f_2(X_i)$ and $f_3(X_i)$. Also, the full expression will not be required if the fanout of x_i is less than the number of terms in $Dz(x_i)$. An alternative therefore is to provide a technique that will generate single terms only, and that may be recycled if more than one term is required. Such a scheme is inherently faster for primary inputs exhibiting little or no fanout, and it is this approach that is suggested for this algorithm.

The mechanism of the process is based on the fact that a term occurring in the s-o-p expression z and containing x_i or $\bar{x}_i$, will be included in $Dz(x_i)$ provided there is no similar term in z_{x_i} to mask it out, i.e., map to the same location in the z_{x_i} Karnaugh map. Those terms in z that do not contain x_i or x_i will obviously be repeated in z_{x_i} and cannot thus appear in $Dz(x_i)$. If a term in z is shown to be a member of $Dz(x_i)$, then that term with the x_i or $\bar{x}_i$ removed, is still a valid member of $Dz(x_i)$—this being a direct result of the independence theorem.

The process therefore is one of selecting a candidate from the expression for z, and exhaustively comparing with *all* terms in z_{x_i} looking for a masking term. If don't cares are initially present in the candidate, they may be used, if necessary, to maintain the validity of the term.

More formally, let the s-o-p expression for z be expressed as:

$$z = t_1 + t_2 + \ldots + t_i + \ldots + t_m \quad (10)$$

where t_i, $1 \leqslant i \leqslant m$ represents the m conjuncts—either in their fully expanded (minterm) form or otherwise.

The process of generating terms in $Dz(x_i)$ is shown in the flow chart of **Fig. 4** and the following comments are pertinent:

1. The complete function z_{x_i} is never formed. Instead single terms only (t_k, $1 \leqslant k \leqslant m$) are generated by selecting from z and inverting the value of x_i if found.
2. To demonstrate the assignment of don't cares, consider the process applied to the s-o-p expression obtained in Table 3, namely:

$$Z1 = X1X2 + \overline{X1}\ \overline{X2} + \overline{X1}\ \overline{X3} \quad (11)$$

In particular, we will select the first term $X1X2$ and attempt to determine its validity in $DZ1(X1)$. The first comparison with $\overline{X1}X2$ contains $X1$ incompatible and we conclude that $X1X2$ is still valid and can remain unaltered. Similarly with the second term $X1\overline{X2}$—$X2$ being incompatible. The third comparison however, with $X1\overline{X3}$ is a 'maybe' since

*An input to a logic gate is said to be at a control status if any change in its value results in a corresponding change at the output of the gate, all other inputs remaining unchanged. If this is not true, the input is said to be static, e.g., for an AND gate having inputs $X1$ $X2$ $X3$

$X1$	$X2$	$X3$	
1	1	1	—all inputs are control
0	1	1	} one control (0), two static
1	0	1	
1	1	0	
0	0	0	—all static
Remainder			—all static unless sensitive paths reconverge on the 0's alone.

$X1\overline{X3}$ will expand to form $X1X2\overline{X3}$ and $X1\overline{X2}\,\overline{X3}$ and the first of these will mask part of the expanded version of $X1X2$. We note, however, that $X3$ is, so far a don't care in the candidate but specified as $\overline{X3}$ in the comparison term. Consequently an addition of $X3$ to the candidate will maintain its validity. In this case, all three terms in $Z1_{X1}$ have failed to mask the candidate—now modified to $X1X2X3$, and we conclude that $X1X2X3$ and hence $X2X3$ is a term in $DZ1(X1)$.

This indicates the disadvantage of the process—namely, what happens if there is more than one don't care to which an assignment can be made to maintain validity? One solution is to only assign one of them in order to leave as many options on future assignments as possible. The danger of course is that the wrong one is chosen, resulting in an invalidation at a later stage. If this happens, it may be possible to return to the selection point and re-assign on another don't care. Currently, the programmed version of this process is capable of two attempts and if validation still fails, the candidate is abandoned.

3. In view of this last point, it may be possible to fail completely in generating a valid term. The author has never experienced this in the program and suspects that its eventuality is so slim as to not merit sophisticated recovery procedures. What is more likely to be the cause of the failure, if it occurs, is the fact that the particular primary input concerned is logically redundant. In view of the fact that absolute minimality of the s-o-p expressions cannot be guaranteed, it is possible for an expression to be formed containing redundancies. If this occurs, then it will be impossible to generate any terms in the Boolean differential relating to this term.

As an example, consider the following s-o-p expression:

$$Z = X1X2 + \overline{X1}\,X3 + \overline{X2}X3 + X3X4 \qquad (12)$$

The absorption checks would be unable to reduce this any further, yet:

$$\begin{aligned} Z &= X1X2 + \overline{X1X2}X3 + X3X4 \\ &= X1X2 + X3, \text{ i.e., independent of } X4 \end{aligned} \qquad (13)$$

It is thus impossible, by any technique, to generate any terms in $DZ(X4)$ from the s-o-p expression in equation (12).

Returning to the main algorithm description, an attempt can be made to generate $p_i\,Dz(x_i)$ terms and hence $2p_i$ tests for each x_i, where p_i is the relevant fanout term. The result of applying this to the example is shown in **Table 4.**

In general, the tests may still contain don't cares and in view of the current random assignment approach, they may be used to advantage to effect a reduction on the number of tests, through judicious merging. One such reduction on Table 4 leads to the following six tests:

$\overline{X1}X2\overline{X3}$, $\overline{X1}X2X3$, $\overline{X1}\,\overline{X2}\,\overline{X3}$, $\overline{X1}\,\overline{X2}X3$, $X1\overline{X2}\,\overline{X3}$, $X1X2X3$

In reality, five of these six tests are sufficient to provide complete detection coverage for this circuit (see Appendix 2), and this indicates the next and penultimate stage in the algorithm—validation or otherwise of the total fault coverage.

Table 4 Tests generated

$X1$	$X2$	$X3$	INPUT
3	2	1	FANOUT
$X1\ \overline{X2}\ \overline{X3}$	$\overline{X1}\ X2\ X3$	$\overline{X1}\ X2\ X3$	TEST PAIRS
$\overline{X1}\ \overline{X2}\ \overline{X3}$	$\overline{X1}\ \overline{X2}\ X3$	$X1\ X2\ \overline{X3}$	
$X1\ \overline{X2}$	$X1\ X2$		
$\overline{X1}\ \overline{X2}$	$X1\ \overline{X2}$		
$X1\ X2\ X3$			
$\overline{X1}\ X2\ X3$			

Stage 3—Validation of the fault-cover

The question we are now posed is—does the test set derived so far cover all single stuck-at faults on both primary and internal logic lines?

There are at least two methods of answering this and the first is to employ the path sensitising concept and, for each test, determine the sensitive paths that are established. This process is referred to as the fault-cover algorithm and is based on identifying the 'control' or 'static' status of each logic line with the defined test input. Sensitive paths may then be constructed by linking together local (gate level) sensitive paths. This process is complicated however by:

1. The need to also identify occurrences of positive and negative reconvergence (defined in Bennetts, 1971) and
2. The 'look-up' table for control and static levels becomes complex for circuits based on elements that are in themselves, more complex than the simple AND, OR, NAND, NOR gates i.e. MSI/LSI elements.

Nevertheless this approach has been successfully programmed for circuits using the simple elements (Baker, 1971) and satisfactory results obtained. It is not proposed at this stage, however, to modify and extend the fault-cover algorithm. Instead, the second alternative would appear to offer a solution to both objections mentioned—this being to use a logic simulator.

By applying the first test to the simulated circuit, and postulating each fault in turn, a partition on the total set of faults can be derived—one subset containing faults detected by the test and the other the remaining undetected faults. Application of the second test on the undetected fault set creates a further subpartition, again into detected and undetected sets. The effects of all subsequent tests is then assessed similarly and the final result is either that the undetected fault set is empty or still contains undetected faults, thereby indicating the inadequacy of the test set. The process can be extended to obtain the locational resolution of the test set (Bennetts, 1971).

The result of either the fault-cover algorithm or logic simulation is to both specify the correct output associated with each test and to indicate any uncovered faults. The final stage therefore is the selective generation of tests for any uncovered faults.

Stage 4—Completion of the test set

The procedure here is very similar to the process already described. Indeed, if the uncovered fault is one of the primary inputs along say, a fanout branch, then the fanout factor may be increased for that input and extra tests generated.

If, however, the uncovered fault occurs internally, then a slightly more sophisticated process must be carried out. The theoretical basis for this is as follows:

Let c_i be an internal connection for which a test is required s-a-1, s-a-0 or both.

$$\text{Let } z = f(X, c_i) \text{ be formed} \qquad (14)$$

In effect, c_i has assumed primary input status.

$$\text{Thus } z_{c_i} = g(X, \bar{c}_i) \text{ is valid and} \qquad (15)$$

$$Dz(c_i) = h(X) \text{ (Note independence of } c_i) \qquad (16)$$

is meaningful and any terms in the s-o-p form of $Dz(c_i)$ have the same significance for c_i as $Dz(x_i)$ has for x_i. Thus $Dz(c_i).c_i$ and $Dz(c_i).\bar{c}_i$ will specify the test conditions for c_i s-a-0 and c_i s-a-1 respectively. The only difference in this case is the need to derive c_i and $\bar{c}_i$ as a function of the primary inputs:

$$c_i = j(X) \qquad (17)$$

The test generation part of the algorithm is therefore modified to:

1. Determine $z = f(X, c_i)$. This is achieved by *not* substituting for c_i and $\bar{c}_i$ in the build-up of the overall Polish form.

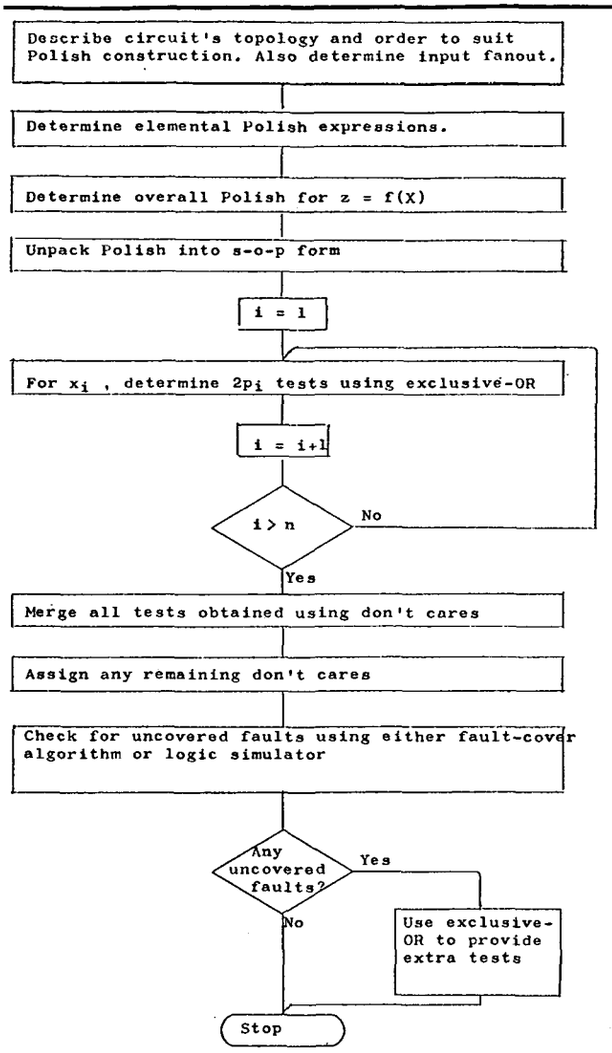

Fig. 5. Flowchart of the complete algorithm

2. Determine $c_i = j(X)$ and $\bar{c}_i = \overline{j(X)}$. Again this is simply achieved by starting with the Polish expression for c_i instead of z, and successively substituting to derive the full Polish for c_i in terms of x's alone. It is worth making the observation here that if this Polish expression for c_i is modified by replacing AND operators with OR operators and vice versa, and also inverting all literals, then it will then represent $\bar{c}_i$ and may be unpacked into its s-o-p form. This is a more efficient process, and uses existing software, than the alternative procedure of applying de Morgan's rule followed by p-o-s to s-o-p expansion, to the c_i s-o-p expression.

It has been found in practice that there is very little demand for this final stage since the tests derived using the fanout heuristic nearly always cover all faults.

Summary of the overall procedure

The overall procedure can now be summarised and is shown diagrammatically in **Fig. 5.** The remaining parts of the paper discuss the extension to multi-output circuits; the use of a logic element equivalent circuit library to allow circuits based on MSI/LSI elements to be processed; the modifications necessary for 'clamped' logic lines and some comment and results relating to the programmed version of the procedure.

Multi-output circuits

The simple answer to multi-output circuits is to treat it in effect as a number of single-output circuits, i.e., repeat the process in Fig. 5 up to, but not including the merging, for each s-o-p expression obtained for each output. The amount of redundancy in such a test set is obviously a function of the amount of interdependence between inputs and outputs. The remainder of the process remains unaltered.

Where the primary inputs exhibit high fanout, then different strategies can be effected, e.g., for an n input, m output circuit having an input x_i possessing a fanout p_i, then if x_i is a member of q s-o-p expressions where $q \leqslant m$, then generation of $2qp_i$ tests will almost certainly contain a high degree of redundancy. One strategy therefore might be to generate $2p_i$ tests from one of the s-o-p expressions and then supplement this by two tests (1 Boolean Differential term) from each remaining $q - 1$ expressions yielding a total of $2(p_i + (q - 1))$ tests.

Use of the logic element equivalent circuit library

Rather than determine the elemental Polish expressions at run time, it is more efficient if these are readily available in a library. The procedure adopted is to input the equivalent circuit of the elements and to pre-process these circuits into their equivalent Polish forms. The equivalent circuits are defined solely in terms of the four basic gates and the same Polish routines are used to build up the overall Polish describing the element proper. This means that tests for circuits based on complex MSI/LSI elements can be generated provided an equivalent circuit has been defined and duly processed. The overall test generation procedure is then identical to simple gate circuits—indeed even these simple gates are held in the library and the process is completely general for simple, complex or 'hybrid' circuits.

Clamped logic lines

Again, in reality, and this applies especially to circuits based on the more complex elements, some of the input connections are clamped at the enable logic level. This has two effects on test generation:

1. It means that no test exists for the logic line stuck-at its clamp level*, and
2. The logical behaviour of the element is simplified.

The first effect can be used to modify the list of uncovered faults, should this list contain any such undetectable 'faults'.

The second effect can be counteracted by an amendment to the elemental Polish expression for the affected element.

E.g. If a circuit contains an element having two 2-input AND gates feeding a 2-input OR gate and described by:

$$C5 = C1C2\ \text{AND(2)}\ C3C4\ \text{AND(2)}\ \text{OR(2)} \rightarrow C1C2 + C3C4$$

and both $C3$ and $C4$ are clamped at logical 0, then the operation of this element is logically reduced to $(C1C2)$ alone and the Polish expression can be modified accordingly to:

$$C5 = C1C2\ \text{AND(2)}$$

The modification process is not quite so simple as it sounds since, if clamped literals are removed, the operator(s) associated with the literals must also be modified. In the case above, two operators were completely removed. Had $C3$ only been clamped (at logical 1 this time), then the Polish is effectively reduced to:

$$C5 = C1C2\ \text{AND(2)}\ C4\ \text{AND(1)}\ \text{OR(2)}$$

The operator count associated with the second AND operator

*In practice, it is relatively easy to identify and reduce occurrences of the following form:

$$X1 + X1\,X2 \rightarrow X1$$
$$X1\,X2 + X1\,X2 \rightarrow X1X2$$
$$X1\,X2 + X1\,\overline{X2} \rightarrow X1$$

It is not feasible, however, to look for occurrences of the type $(X1 + \overline{X1}X2)$ although this would reduce to $(X1 + X2)$.

has been reduced by 1—in fact, if the count is ever reduced to 1, the operator itself can be removed. There is no difference between the expression above with AND(1) in and the same expression with AND(1) removed. This rule is quite general provided only AND or OR operators are used.

Comment and results of the program

The procedure described in Fig. 5 has been programmed and all the features so far mentioned have been included—namely multi-output circuit facilities, library of equivalent circuits and clamped logic lines. The program has been written for an ICL 1907 computer and the language is mainly FORTRAN with a number of PLAN subroutines.

Extensive use has been made of a List Processing language previously developed at Southampton University (Waters, 1970). This has been for a number of reasons, the main one being that this program is to be compatible with and eventually embedded into, the logic synthesis program suite developed by Professor D. W. Lewin (now at Brunel University), and his research team (Lewin, Purslow, and Bennetts, 1972). Another major advantage of a List Processor is that nearly all the operations necessary to perform the overall procedure are amenable to a list format and symbol manipulative problems of this nature are best solved, on a numerical computer, by adopting such a list data structure.

One of the disadvantages of the List Processor, as it stands, is that it places an artificial core store restriction of 32K on the program. This in turn limits the size of circuit that can be processed. The limit however, is related, not so much to the number of elements, inputs or outputs, but to the size of the s-o-p expressions. Once this has been determined, subsequent core store requirements are fairly trivial. At present, the program will exceed available core store if it attempts to generate a s-o-p expression containing more than 3,000 terms. It can be seen therefore that apart from any theoretical considerations, the minimality or near-minimality of the s-o-p expressions is vital and the absorption checking procedures are extremely important in this respect.

A number of circuits of different types of complexity have been successfully handled by the program and brief details of the results obtained, are presented in **Table 5.**

These results are not really very significant without the actual circuit diagrams, but they do indicate to some degree the capability of the program, working within the 32K core store bound.

Concluding remarks

It is felt that the procedure that has been described and programmed for generating detection test sets for combinational logic circuits, is a realistic approach to solving this problem. There have been many algorithms described in the literature (see Bennetts and Lewin (1971) for instance), most of them theoretically sound, but problems arise when attempts are made to put such algorithms into practice.

The algorithm that has just been described is really an amalgamation of some of the better aspects of these various separate algorithms, together with some theoretical modification and engineering compromise. It is not offered as yet another approach to deriving test sequences—rather, it is offered as a realistic technique that has been proved in practice.

Acknowledgement

The author would like to acknowledge the effort of J. D. Baker in his programming of the fault-cover algorithm; the assistance of J. L. Washington in the development of the main program and the continual support and encouragement of his project supervisor, Professor D. W. Lewin (now of Brunel University).

He would also like to acknowledge the financial support of SRC and ICL for an industrial studentship.

Appendix 1

Proof of independence of $Dz(x_i)$ on x_i

The following theorem is quoted (incorrectly), but not proved, in (Sellers, Hsiao, and Bearnson, 1968b).

Theorem

Given

$$z = f(x) = x_i f_1(X_i) + \bar{x}_i f_2(X_i) + f_3(X_i)$$

then

$$Dz(x_i) = \overline{f_3(X_i)}\,[f_1(X_i) \oplus f_2(X_i)]$$

Table 5 Some results of the program

EXAMPLE NUMBER	1	2	3	4	5
Number of inputs/outputs	3/1	4/1	15/1	11/4	55/7
MSI/LSI based?	No	No	No	No	Yes
Number of basic gates	5	8	21	62	95
Primary input fanout?	Yes	Yes	Yes	Yes	Yes
Number of terms in s-o-p expressions	3	2	64	49, 21, 16, 6	30, 332, 392, 4, 4, 4, 119
Time to derive s-o-p expressions (secs.)	2	3	30	37	1082
Total number of tests required using fanout heuristic	12	20	54	96	228
Total number of tests generated	12	16	54	96	228
Number of tests after merging	6	10	34	22	65
% reduction	50%	37%	37%	71%	82%
Time to generate and merge tests (secs.)	1	1	17	25	252
Time to calculate output associated with each test (secs.)	1	1	2	3	58
Maximum core store used	9K	9K	12K	14K	26K

Example 1: Example used in paper
2: Schneider's circuit
3: Amar and Condulmari's circuit
4: Four bit parallel adder
5: 'Concocted' larger circuit using MSI elements

where

$$X_i = \{x_1, x_2, \ldots, x_{i-1}, x_{+1}, \ldots, x_n\}$$

Proof
Given

$$z = x_i f_1(X_i) + \bar{x}_i f_2(x_i) + f_3(X_i)$$

From equation (2) we have:

$$z_{x_i} = \bar{x}_i f_1(X_i) + x_i f_2(X_i) + f_3(X_i)$$

and therefore from equation (3) we have:

$$Dz(x_i) = [x_i f_1(X_i) + \bar{x}_i f_2(X_i) + f_3(X_i)] \oplus [\bar{x}_i f_1(X_i) + x_i f_2(X_i) + f_3(X_i)]$$

For ease of notation, we will represent x_i by x, $f_1(X_i)$ by f_1, $f_2(X_i)$ by f_2 and $f_3(X_i)$ by f_3.

Therefore:

$$\begin{aligned}
Dz(x_i) &= [xf_1 + \bar{x}f_2 + f_3] \oplus [\bar{x}f_1 + xf_2 + f_3] \\
&= [xf_1 + \bar{x}f_2 + f_3]\,\overline{[\bar{x}f_1 + xf_2 + f_3]} \\
&\quad + \overline{[xf_1 + \bar{x}f_2 + f_3]}\,[\bar{x}f_1 + xf_2 + f_3] \\
&= [xf_1 + \bar{x}f_2 + f_3]\,[(x + \bar{f}_1)(\bar{x} + \bar{f}_2)(\bar{f}_3)] \\
&\quad + [\bar{x}\bar{f}_1 + xf_2 + f_3]\,[(\bar{x} + \bar{f}_1)(x + \bar{f}_2)(\bar{f}_3)] \\
&= \bar{f}_3[xf_1 + \bar{x}f_2 + f_3]\,[\bar{x}\bar{f}_1 + x\bar{f}_2 + \bar{f}_1\bar{f}_2] \\
&\quad + \bar{f}_3[\bar{x}f_1 + xf_2 + f_3]\,[x\bar{f}_1 + \bar{x}\bar{f}_2 + \bar{f}_1\bar{f}_2] \\
&= \bar{f}_3[\bar{x}\bar{f}_1 f_2 + xf_1\bar{f}_2] + \bar{f}_3[x\bar{f}_1 f_2 + \bar{x}f_1\bar{f}_2] \\
&= \bar{f}_3[(x + \bar{x})(f_1\bar{f}_2 + \bar{f}_1 f_2)] \\
&= \bar{f}_3(f_1 \oplus f_2)
\end{aligned}$$

i.e.

$$Dz(x_i) = \overline{f_3(X_i)}\,[f_1(X_i) \oplus f_2(X_i)] \qquad \text{Q.E.D.}$$

We note in passing the following two corollaries.

Corollary 1
Given

$$z = x_i f_1(X_i) + \bar{x}_i f_2(X_i)$$

Then

$$Dz(x_i) = f_1(X_i) \oplus f_2(X_i)$$

Proof
This is the special case of the theorem in which

$$f_3(X_i) = 0 \text{ i.e. } \overline{f_3(X_i)} = 1$$

Corollary 2
Given

$$z = x_i f_1(X_i)$$

Then

$$Dz(x_i) = f_1(X_i)$$

Proof
Suppose $f_2(X_i) = 0$. Then, from corollary 1

$$\begin{aligned}
Dz(x_i) &= f_1(X_i) \oplus 0 \\
&= f_1(X_i).1 + 0.\overline{f_1(X_i)} \\
&= f_1(X_i)
\end{aligned}$$

The significance of the theorem and its corollaries is that if $z = f(X)$ and $x_i \in \{X\}$, *then* $Dz(x_i)$ *is independent of* x_i.

Appendix 2

Use of the exclusive-OR operator to accommodate primary input lines possessing fanout

It was stated in the paper that if a primary input x_i had a fanout p_i, then it was a useful heuristic to generate $2p_i$ tests by deriving p_i terms from the Boolean differential $Dz(x_i)$. The basis for this was that if the p_i terms established p_i sensitive paths that exercised all the gates to which x_i was an input, then it was probable that most, if not all, internal lines logically dependent on x_i would also be covered. In practice however, it was pointed out that parallel sensitive paths are an unknown factor. Also, unless the full expression for $Dz(x_i)$ were known, there was no guarantee that the correct three terms had been selected. If the correct three terms were selected however, the probability of complete fault coverage increases.

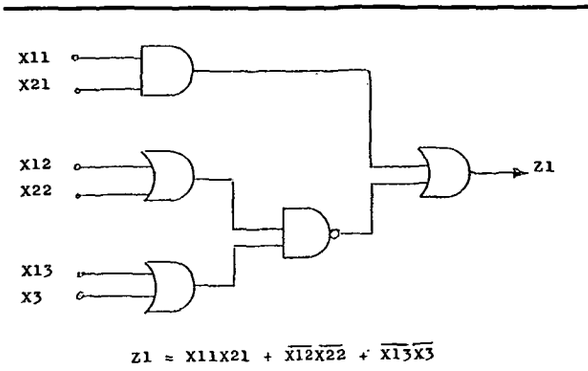

Fig. 6. Example with separate input identification

This appendix offers a theoretical solution to ensuring that such terms are always selected and is based on a separate identification of any primary input line that possesses fanout. The bulk of the procedure is then unaffected until the actual term is generated. Here, certain constraints must be applied to ensure that the separate lines are not specified with an incompatible polarity requirement.

As an example, we will consider the same example of Fig. 1 but with $X1$ replaced by $X11$, $X12$, $X13$ and $X2$ by $X21$, $X22$. The circuit is redrawn in **Fig. 6.**

Consider now the expression obtained for $DZ1(X11)$:

$$\begin{aligned}
DZ1(X11) &= [X11\,X21 + \overline{X12}\,\overline{X22} + \overline{X13}\,\overline{X3}] \oplus [\overline{X11}\,X21 + \overline{X12}\,\overline{X22} + \overline{X13}\,\overline{X3}] \\
&= X21(X12 + X22)(X13 + X3) \qquad (18)
\end{aligned}$$

As usual, this expression is independent of $X11$ and the full test set for $X11$ is given by:

$$DZ1(X11).(X11 + \overline{X11}) = [X21(X12 + X22)(X13 + X3)]\,[X11 + \overline{X11}] \qquad (19)$$

Expansion of equation (19) produces:

$$\begin{aligned}
DZ1(X11).(X11 + \overline{X11}) = {} & X12\,X13\,X21\,X11 + X12\,X13\,X21\,\overline{X11} \\
& + X12\,X3\,X21\,X11 + X12\,X3\,X21\,\overline{X11} \\
& + X22\,X13\,X21\,X11 + X22\,X13\,X21\,\overline{X11} \\
& + X22\,X3\,X21\,X11 + X22\,X3\,X21\,\overline{X11} \qquad (20)
\end{aligned}$$

An inspection of the terms in equation (20) reveals that there are three that are $X1$-polarity incompatible. The equation must therefore be modified, and subsequent replacement of $X11$, $X12$, $X13$ by $X1$ etc. produces:

$$DZ1(X1).(X11 + \overline{X11}) = X1\,X2 + \overline{X11}\,X2\,X3 \qquad (21)$$

Equation (21) therefore represents three tests, two for $X11$ s-a-0 and one for $X11$ s-a-1, that guarantee to exercise at least the gate to which $X11$ is an input, i.e. $E1$.

Similar expressions for the remaining five inputs may be derived and this leads to the shortened (primary input faults only) fault matrix shown in **Table 6.** A minimal cover obtained from this matrix is given by:

$$\text{Test set} = t_1\,t_2\,t_3\,t_4\,t_7 \qquad (22)$$

and in practice, this test set will cover all internal faults.

One final comment in connection with this. It would be a false assumption to say that if, say the s-a-0 tests for $X11$, $X12$ and $X13$ were collected together, they would equal the s-a-0 tests for $X1$ that were derived from $DZ1(X1)$. The validity of the tests in Table 6 rests on the implicit assumption that only one of the fanout branches is faulty—all other branches must be logically correct in order to establish the necessary sensitive

Table 6 Shortened fault matrix

	$X1$	$X2$	$X3$	$X11$		$X12$		$X13$		$X21$		$X22$		$X3$	
				$s0$	$s1$	$s0$	$s1$	$s0$	$s1$	$s0$	$s1$	$s0$	$s1$	$s0$	$s1$
t_0	0	0	0												
t_1	0	0	1				√						√		
t_2	0	1	0						√						√
t_3	0	1	1		√							√		√	
t_4	1	0	0			√		√			√				
t_5	1	0	1			√					√				
t_6	1	1	0	√						√					
t_7	1	1	1	√						√					

$s0$ = s-a-0, $s1$ = s-a-1

path.* Strictly speaking therefore, the shortened fault matrix should also include the s-a-0, s-a-1 columns for $X1$ and $X2$ respectively, in which the tests have been derived from $DZ1(X1)$ and $DZ1(X2)$. In this case, the tests specified in equation (22) will also cover the $X1$ and $X2$ faults.

*This is not true for its inverse level. Despite the fact that it cannot be set-up, all other lines presumably can and this may be used to test the inverse fault. In a sense, if it is shown that it is not stuck-at its inverse level, then it has been tested for its stuck-at-clamp level.

References

BAKER, J. D. (1971). The fault-cover problem in combinational logic circuits, M.Sc. Thesis, Dept. of Electronics, University of Southampton.

BENNETTS, R. G. (1971). The diagnosis of logical faults, *Wireless World*, Part 1, July 1971, pp. 325-328, Part II, August 1971, pp. 383-385, letter to Editor Sept. 1971, pp. 428-429.

BENNETTS, R. G., and LEWIN, D. W. (1971). Fault diagnosis of digital systems—a review, *The Computer Journal*, Vol. 14, No. 2, pp. 199-206. Also *IEEE Computer*, July/Aug. 1971, pp. 12-20.

LEWIN, D. W., PURSLOW, E., and BENNETTS, R. G. (1972). Computer assisted logic design—the CALD system, IEE Conference Publication No. 86, CAD conference, University of Southampton, pp. 343-351.

ROTH, J. P. (1966). Diagnosis of automata failure: a calculus and a method, *IBM Journal R & D*, Vol. 10, pp. 278-291.

SELLERS, F. F., HSIAO, M. Y., and BEARNSON, L. W. (1968a). Analysing errors with the Boolean difference, *IEEE Trans. on Computers*, Vol. C-17, pp. 676-683.

SELLERS, F. F., HSIAO, M. Y., and BEARNSON, L. W. (1968b). Error detecting logic for digital computers, p. 25. McGraw-Hill.

WATERS, M. C. (1970). A list-processing language for use in FORTRAN and ALGOL, Internal report, Dept. of Electronics, University of Southampton.

Correspondence

To the Editor
The Computer Journal

Sir,

Permit me to draw your attention to some errors in the article 'A quasi-intrinsic scheme for passing a smooth curve through a discrete set of points' in your issue of November 1970.

The equations (1) should give $\frac{dy}{dt}$ as

$$\frac{dy}{dt} = \sin\theta_K + (\sin\theta_{K+1} - \sin\theta_K)\frac{t}{T} + C_K t(T - t)$$

The equation as given is therefore wrong by reason of symmetry and the fact that both $\frac{dx}{dt}$ and $\frac{dy}{dt}$ are stated to be quadratics in t (this is only true of $\frac{dx}{dt}$ as given in the article).

Upon integration of the corrected equations (1) to give equations (2), the equation for y should be

$$y = y_K + \sin\theta_K t + (\sin\theta_{K+1} - \sin\theta_K)\frac{t^2}{2T} + C_K\left(\frac{Tt^2}{2} - \frac{t^3}{3}\right)$$

I have programmed both the printed version and my corrected version and have found that only the corrected version gives P_K for $t = 0$ and P_{K+1} for $t = T > 0$ (where T has the value T_K as given elsewhere in the article).

Yours faithfully,

W. COOPER (student)

Glasgow University Computing Dept.
Glasgow W.1

23 February 1972