A Method to Calculate Necessary Assignments in Algorithmic Test Pattern Generation

Janusz Rajski and Henry Cox

Department of Electrical Engineering
McGill University, 3480 University Street
Montréal, Canada H3A 2A7

Abstract

Necessary assignments are those which must be made in order to find a test pattern. This paper presents new algorithms based on the concept of *reduction lists* to determine necessary assignments. The algorithms use a 16-valued logic system and are based on the mathematical concepts of images and inverse images of set functions.

Experimental results are presented for a variety of benchmark circuits.

1. Introduction

Test pattern generation can be viewed as a branch and bound problem [8]: test pattern generation algorithms usually search through the space by systematically branching and bounding until either a test pattern is discovered or the search space is exhausted. Along the way, certain assignments can be distinguished as being *necessary*: viewed as a branch decision, assigning them to some other value guarantees that a bound step must eventually be taken. That is, if the necessary assignments are not made, then the subspace which has been branched into is guaranteed *not* to contain a test pattern. Thus, alternative choices for necessary assignments need not be explored. Other assignments can be distinguished as *nonconflicting* in that they lead in the direction of a test and restrict the search space, but never need to be backtracked. The remaining assignments are *arbitrary*—they may or may not lead in the direction of a test, and may or may not need to be backtracked. This paper addresses the problem of identifying necessary assignments.

The PODEM [8] and FAN [7] algorithms identify some necessary assignments using local implications, but rely on heuristics for choosing "good" branch nodes (arbitrary assignments) for much of their power. The contribution of SOCRATES [14] was to find additional necessary assignments which could not be found using local implication.

SOCRATES determines additional necessary assignments by finding the effect of each possible assignment to every node in the circuit. If an assignment makes it impossible to achieve some required value, then that assignment must be disallowed. The effect of a particular assignment is determined by injecting the logic value in the circuit and determining its implications. This technique does not take logic dependencies between circuit nodes into account, and thus does not identify all necessary assignments. To overcome this problem, common logic modules (adders, multiplexors, *etc.*) whose logic dependencies are predetermined, have been added to the library of building blocks recognized by a modular version of SOCRATES [13]. Before each new module can be recognized, implication, unique sensitization, and multiple backtrace procedures which take the signal dependencies of the module into account must be manually determined and added to the system. Dependencies between modules and in unrecognized structures continue to be missed.

In this paper, we present a test pattern generation algorithm which finds all necessary assignments, including those which arise due to logic dependencies between circuit nodes. The algorithm is based on the mathematical concept of images and inverse images of set functions. We generalize and formalize the process of necessary assignment extraction using the idea of *reduction lists*, and show that both classical implication and "learning" [14] are special cases of a more general technique. In order to take advantage of formal concepts developed for Boolean algebras, the algorithm employs a 16-valued algebra for test pattern generation. We illustrate the benefits of a 16-valued system through examples of faults which are not properly handled by conventional 5 or 9-valued systems.

In the test pattern generation algorithm, necessary and nonconflicting assignments are extracted iteratively until the fault is either tested, proven to be redundant, or until no more assignments can be found, at which point an arbitrary assignment (branch decision) is made. Experimental results show that many faults are tested or proven to be redundant without branching.

This paper is divided into three sections. First, we introduce the 16-valued logic system which is the foundation of our approach. Next, we introduce the concepts of necessary assignments and reduction lists and discuss their use in test pattern generation. Finally, we present experimental results obtained by our test pattern generation system when run on

This work was supported by strategic grant MEF0045788 from the Natural Sciences and Engineering Research Council of Canada.

CH2910-6/0000/0025$01.00 © 1990 IEEE

a variety of benchmark circuits.

2. The Alphabet

The two-element Boolean algebra $B_2^1 = \{0,1\}$ is widely used to analyse switching circuits. It is also sufficiently precise to describe the behaviour of a fault-free combinational circuit. However, in order to describe the behavior of a possibly faulty circuit, a four-element Boolean algebra, $B_2^2 = \{0(0), 0(1), 1(0), 1(1)\}$, where $a(b)$ indicates that the response in the fault-free circuit is a and in the faulty circuit is b, is required. Using the D-symbols, $B_2^2 = \{0, \overline{D}, D, 1\}$. The function of a two-input gate is described as a mapping $B^2 \times B^2 \rightarrow B_2^2$.

Given a test vector for a particular fault, each line in the circuit will carry one of the four possible values from B_2^2. When we set out to find a test pattern, we do not know the actual values taken by each line in the final test vector; thus, we start the process by assigning to each line the *set* of possible values which it could take in *any* test pattern. As test pattern generation proceeds, we determine that certain of the values in each set cannot actually be obtained, and the sets of possible values for each line become more and more refined. For example, if a particular line cannot be affected by the fault(s), its value cannot be either D or $\overline{D}$, and thus its set of possible values is reduced to $\{0,1\}$. Any of the 16 subsets of the set $\{0, 1, D, \overline{D}\}$ is a possible assignment; therefore, a complete alphabet contains 16 values.

B_{16}	$P(B_2^2)$	B_2^4
		$x_1 x_D x_{\overline{D}} x_0$
0	$\{\}$	0000
1	$\{0\}$	0001
2	$\{\overline{D}\}$	0010
3	$\{0, \overline{D}\}$	0011
4	$\{D\}$	0100
5	$\{0, D\}$	0101
6	$\{\overline{D}, D\}$	0110
7	$\{0, \overline{D}, D\}$	0111
8	$\{1\}$	1000
9	$\{0, 1\}$	1001
10	$\{\overline{D}, 1\}$	1010
11	$\{0, \overline{D}, 1\}$	1011
12	$\{D, 1\}$	1100
13	$\{0, D, 1\}$	1101
14	$\{\overline{D}, D, 1\}$	1110
15	$\{0, \overline{D}, D, 1\}$	1111

Table 1 Three codings of a 16-element alphabet

Since we use the subsets of B_2^2 to represent the sets of possible values at each point in the circuit at various stages of test pattern generation, it is natural to introduce the power set $P(B_2^2)$ of B_2^2. The power set of the basic D symbols has been used by Akers for test generation [1]; as it is a Boolean algebra, it is isomorphic to the 16-valued system used in [5, 12] for fault diagnosis. $P(B_2^2)$ has 16 elements, and can be coded by natural numbers from 0 to 15 (B_{16} in Table 1), or as bitwise encoded quadruples describing the presence or absence of elements of B_2^2 (B_2^4 in Table 1).

Various algebras for test pattern generation have been proposed. Use of an appropriate algebra can greatly aid in test pattern generation. Comparisons between algebras typically focus on the number of elements each contains, the space required to store circuit values, and the time required to manipulate them [4]. A better comparison is the ability of the logic system to resolve circuit values during test pattern generation. Better resolution of values may allow a test pattern generation algorithm to reduce the amount of branching and backtracking which must be performed to find a test or prove a fault redundant, thus requiring less time and storage space despite using an alphabet which contains more values.

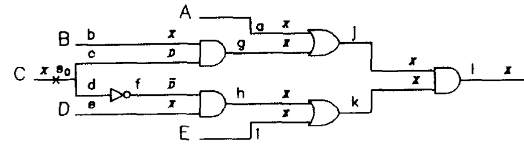


Figure 1 Test pattern generation using a 5-valued alphabet

The 5-valued alphabet $A_5 = \{0, 1, D, \overline{D}, X\}$, where X indicates "unknown" and the other symbols are interpreted as in B_2^2, has been used in many ATPG algorithms [7, 8, 14]. When *targeting* (attempting to generate a test for) fault C/s_0, as shown in Fig. 1, we quickly determine that most circuit values are X, and are forced to make several arbitrary branch decisions before eventually finding a test vector.

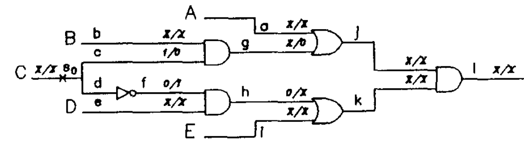


Figure 2 Test pattern generation using a 9-valued alphabet

Algorithms employing a 9-valued alphabet $A_9 = \{0/0, 1/1, 1/0, 0/1, X/0, 0/X, 1/X, X/1, X/X\}$, where a/b indicates the value in the fault-free/faulty circuit and X indicates unknown [3, 9, 11], encounter a similar problem (Fig. 2), although the additional values may aid in the choice of good branch decisions.

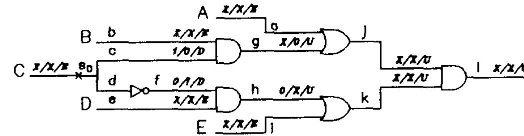


Figure 3 Test pattern generation using the SPLIT circuit model

In the SPLIT model for test pattern generation [4], values in the good and faulty machines are treated separately—each

value can be 0, 1, or X, independent of the value in the other machine. In addition, the relation between the values in the good and faulty machine is calculated. Signal values in the split model can be identified by the triple $G/F/R$, where G is the value in the good machine, F is the value in the faulty machine, and R is the relation between the values (Equivalence, Difference, or Unknown). Thus, circuit values in the SPLIT model are taken from the set $A_{11} = \{0/0/E, 1/1/E, X/X/E, 1/0/D, 0/1/D, X/X/D, 0/X/U, X/0/U, 1/X/U, X/1/U, X/X/U\}$. Compared to the 9-valued system, the SPLIT model is able to distinguish the values $\{0,1\}$ and $\{D,\overline{D}\}$—the values in the good and faulty machines are both X but are closely correlated. However, the SPLIT model is unable to distinguish the values $\{0,1,D\}$, $\{0,1,\overline{D}\}$, $\{0,D,\overline{D}\}$, $\{D,\overline{D},1\}$, and $\{0,1,D,\overline{D}\}$, all of which are represented by $X/X/U$, and so fails to determine a test in the example shown in Fig. 3.

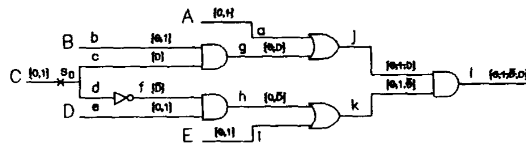


Figure 4 Test pattern generation using a 16-valued alphabet

Using a 16-valued alphabet [1, 12], we recognize that in order to observe $\{D\}$ at the output, assignments $A = \{0\}$, $B = C = \{1\}$ are required (Fig. 4). Similarly, in order to observe $\{\overline{D}\}$ at the output, assignments $A = D = \{1\}$, $E = \{0\}$ are required. In both cases, a test is found immediately with no arbitrary assignments required.

B_{16}	$P(B_2^2)$	A_{11}	A_9	A_5
0	$\{\}$	—	—	—
1	$\{0\}$	$0/0/E$	$0/0$	0
2	$\{\overline{D}\}$	$0/1/D$	$0/1$	$\overline{D}$
3	$\{0,\overline{D}\}$	$0/X/U$	$0/X$	X^*
4	$\{D\}$	$1/0/D$	$1/0$	D
5	$\{0,D\}$	$X/0/U$	$X/0$	X^*
6	$\{\overline{D},D\}$	$X/X/D$	X/X^*	X^*
7	$\{0,\overline{D},D\}$	$X/X/U^*$	X/X^*	X^*
8	$\{1\}$	$1/1/E$	$1/1$	1
9	$\{0,1\}$	$X/X/E$	X/X^*	X^*
10	$\{\overline{D},1\}$	$X/1/U$	$X/1$	X^*
11	$\{0,\overline{D},1\}$	$X/X/U^*$	X/X^*	X^*
12	$\{D,1\}$	$1/X/U$	$1/X$	X^*
13	$\{0,D,1\}$	$X/X/U^*$	X/X^*	X^*
14	$\{\overline{D},D,1\}$	$X/X/U^*$	X/X^*	X^*
15	$\{0,\overline{D},D,1\}$	$X/X/U^*$	X/X^*	X^*

* indicates values which cannot be distinguished

Table 2 Comparison between algebras

Table 2 compares the values represented by a 16-element alphabet [1, 12] with those of the SPLIT, [4], 9-valued [11], and 5-valued alphabets. For example, there are five elements of the 16-valued alphabet which cannot be distinguished by the SPLIT model—they are all represented by $X/X/U$; the situation is progressively worse if a 9 or 5-valued alphabet are used. The inability to distinguish circuit values may lead to unnecessary branching and bounding. Note that element "0" ($\{\}$) of the 16-valued alphabet indicates inconsistency (no test pattern exists), and has no representation in any of the other algebras.

A major advantage of increased resolution is that it is possible to determine necessary assignments in the region reached by the fault effect, which is not possible using a 5 or 9-valued alphabet. The use of a 16-valued alphabet has a number of other advantages in addition to increased value resolution. A test pattern generation algorithm using a 16-valued alphabet need not perform "D-drive" or "X-path check" operations and need not maintain a "D-frontier". Value justification is the only operation required by the algorithm. Forward propagation determines all possible values which could be carried by each line in the circuit, including lines the region reached by the fault effect. Thus, the set of outputs to which the fault effect may propagate is known (those outputs whose set of possible values includes D and/or $\overline{D}$); the test generation process begins with the initial set of *justification points* (set of node/value combinations which must be justified) that the fault must be sensitized—the point of the fault must be driven to a value opposite that caused by the fault—and the fault effect must propagate to at least one primary output—either D or $\overline{D}$ must be observed at some output.

3. Images and inverse images of sets

During *forward propagation*, we determine the set of possible values at the output of each gate given the sets of possible values at its inputs. We assume that the values at the inputs to the gate are independent—thus the possible output values are simply those which can be produced by each of the possible combinations of input values.

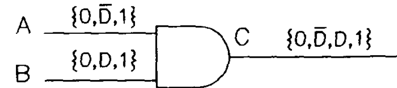


Figure 5 Images for a 2-input *AND* gate

Example 1: Consider the two-input *AND* gate shown in Fig. 5. The sets of possible values on the inputs are $\{0,\overline{D},1\}$ and $\{0,D,1\}$. Thus the set of possible values at the output is:

$$\begin{aligned} AND(\{0,\overline{D},D\},\{0,D,1\}) = \{&AND(0,0), AND(0,D),\\ &AND(0,1), AND(\overline{D},0),\\ &AND(\overline{D},D), AND(\overline{D},1),\\ &AND(1,0), AND(1,D),\\ &AND(1,1)\}\\ = \{&0,\overline{D},D,1\} \end{aligned}$$

This calculation can be formalized using the concept of images of set functions [12]:

Definition 1: Let f be a function of two variables, and let A and B be subsets of $P(B_2^2)$. The image $f(A,B)$ of $A \times B$ under f is the set of all images $f(x,y)$ such that $x \in A$ and $y \in B$. Using set builder notation:

$$f(A,B) = \{ f(x,y) \mid x \in A \text{ and } y \in B \}$$

Using the bitwise encoding B_2^4 from Table 1, the function of a gate can be described by four characteristic equations. The equations determine the presence or absence of each possible value at the output of a gate given the possible values of its inputs. The characteristic equations of an *AND* gate with inputs A and B and output C are:

$$\begin{aligned} c_0 &= a_0 + b_0 + a_D b_{\overline{D}} + a_{\overline{D}} b_D \\ c_{\overline{D}} &= a_1 b_{\overline{D}} + a_{\overline{D}} b_1 + a_{\overline{D}} b_{\overline{D}} \\ c_D &= a_1 b_D + a_D b_1 + a_D b_D \\ c_1 &= a_1 b_1 . \end{aligned}$$

For example, the equation for c_1 says that 1 is a possible value at the output of an *AND* gate only if 1 is a possible value of both inputs. Similar equations can be defined for *OR*, *XOR*, *etc.* gates (as well as for larger functional blocks, if desired). For each gate type, a table can be precomputed which gives the image at the output for each combination of values at the input of the gate, as was done in [5].

Another operation which is required during test pattern generation is *backward implication*, where we determine the smallest set of values at the inputs of a gate which could be combined to produce a restricted set of values at the output of the gate—the inverse of the image function just described.

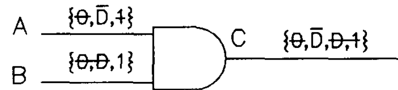


Figure 6 Inverse images for a 2-input *AND* gate

Example 2: Consider the *AND* gate in Fig. 6. If output value $\{\overline{D}\}$ is desired (signified by crossing out the other possible output values we do not want), then the value of input A must be $\{\overline{D}\}$ and of input B must be $\{1\}$. If either input carried some other value, then the set of possible values at the output would be different.

This can be formalized using the concept of inverse images of set functions [12]:

Definition 2: Let f be a function of two variables, and A, B, and C be nonempty subsets of $P(B_2^2)$. The inverse image of c on coordinate X under f, restricted to $A \times B$, which we denote $f^{-X}_{|A\times B}(C)$, is the set of all $x \in A$ such that $f(x,y) \in C$ for some $y \in B$. In set builder notation:

$$f^{-X}_{|A\times B}(C) = \{ x \in A \mid f(x,y) \in C \text{ for some } y \in B \}$$

$$f^{-Y}_{|A\times B}(C) = \{ y \in B \mid f(x,y) \in C \text{ for some } x \in A \}$$

Using the bitwise encoding B_2^4, the inverse image for a gate can also be described by four characteristic equations, as was done for images. For an *AND* gate with inputs A and B and output C, the inverse image A' on input A of set C' is:

$$\begin{aligned} a'_0 &= a_0 c'_0 \\ a'_{\overline{D}} &= a_{\overline{D}}(b_1 c'_{\overline{D}} + b_D c'_0 + b_{\overline{D}} c'_{\overline{D}} + b_0 c_0) \\ a'_D &= a_D(b_1 c'_D + b_D c'_D + b_{\overline{D}} c'_0 + b_0 c_0) \\ a'_1 &= a_1(b_1 c'_1 + b_D c'_D + b_{\overline{D}} c'_{\overline{D}} + b_0 c_0). \end{aligned}$$

Note that the inverse image on input A can be written as the intersection of the current value of A with the generalized inverse image on $\{0, \overline{D}, D, 1\}$ of C' and input B. The inverse image operation can also be performed by table look up [5].

The definition of the image and inverse image set functions for a gate assume that the gate's inputs are independent. However, the input values may be correlated due to reconvergent fanout. This may cause some pessimism in the calculation of images and inverse images, as not all the values in the sets (and, in particular, not all the combinations of values) may actually be obtainable. A method to eliminate this pessimism is described in section 4.3.

4. Reduction Lists and Necessary Assignments

The process of test pattern generation is one of progressively translating a set of required values at some nodes in the circuit to a new set of requirements at other nodes which satisfy the original requirements, but are closer to primary inputs. A test pattern is generated when the required values are completely translated to assignments on primary inputs; the fault is redundant if it is not possible to justify the values.

At any point during test pattern generation, the state of the process is represented by a set of justification points. Given a set of justification points, others can be derived in two ways:

- If an assignment to a particular node leads to a conflict, then it is mandatory that the node be assigned to its alternate value(s). If the set of alternate values is empty ($\{\}$), then no test pattern exists in the space defined by current assignments. The fault is redundant if there are no more arbitrary assignments which can be reversed.
- We arbitrarily decide that we will search the tree in a particular direction, and assign a *branch node* to a particular value. Note that the decision may not be correct and that this decision may be reversed later on.

The set of justification points can be represented by an *AND-OR* graph, where the *and*-nodes represent assignments all of which must be justified in order to find a test and the *or*-nodes represent assignments at least one of which must be justified. For example, in order to generate a test for a fault, the point of the fault must be driven to a value opposite that caused by the fault and D or $\overline{D}$ must be observed on at least one primary output.

Our goal is to find all assignments which are *necessary* in the sense that if we were to assign their value differently,

then the required value of some justification point would no longer be satisfiable. Viewed as a branch decision, making a *reduction assignment* is equivalent to branching into an area of the search space which is guaranteed not to contain a test vector.

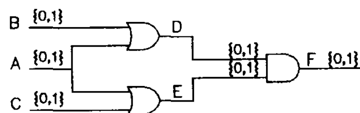


Figure 7 Test pattern generation for F s_1

Example 3: In order to test a s_1 fault on line F of the circuit shown in Fig. 7, the output of the *AND* gate must be $\{0\}$; since F is a primary output of the circuit, propagating the fault effect is trivial. Note that if input A were assigned to $\{1\}$, then the value of both lines D and E would be $\{1\}$—thus, the *AND* gate output would be $\{1\}$, and we would be unable to test the fault. Node A assigned to $\{1\}$ is a reduction assignment; a necessary assignment (and a second justification point) in this example is node A assigned to value $\{0\}$.

4.1 Reduction lists and assignment propagation

In this section, we present a, systematic method of determining necessary assignments through the calculation of *reduction lists*. For each possible value of each line in the circuit, the corresponding reduction list gives the set of reduction assignments for that line and value.

Definition 3: For each line l in the circuit and each possible value v which it could take, the reduction list R_v^l contains those assignments to nodes of the circuit which would cause value v to vanish from the set of possible values of l.

An assignment is a pair consisting of a node identifier and a value (from $P(B_2^2)$). For example, R_0^F for the circuit in Fig. 7 contains the assignment $A_{\{1\}}$ (read "node A assigned to value $\{1\}$"), since assigning A to $\{1\}$ causes 0 to vanish from the set of possible values at F. In other words, $A_{\{1\}}$ *reduces* $F_{\{0\}}$. If $F_{\{0\}}$ is a requirement, then assignment $A_{\{1\}}$ must be *eliminated*—node A must be assigned to whichever values remain after $\{1\}$ has been removed from its set of possible values.

Necessary assignments are derived from the reduction lists at the justification points. If $C_{\{z\}}$ must be justified, then all assignments which appear on reduction list R_z^C must be eliminated. That is, if $P_{\{v\}}$ is an assignment which would reduce $C_{\{z\}}$, then value v must be removed from the set of possible values of point P. If the set of values at P becomes empty, then there is a conflict, and we must backtrack.

In order to eliminate a value from the output of a gate, it is necessary to eliminate all combinations of input values which give rise to that output value. For example, from the characteristic equations for a two-input *AND* gate given in section 3, value 0 is included in the set of possible values at the output of the gate if 0 is present at either input, or if D at one input can be combined with $\overline{D}$ at the other. Thus, in order for an assignment to eliminate 0 at the output of the gate, it must reduce: 0 at both inputs, either D at input A or $\overline{D}$ at input B, and either $\overline{D}$ at input A or D at input B. In other words, an assignment must appear on $\overrightarrow{R}^A_0$, $\overrightarrow{R}^B_0$, $\overrightarrow{R}^A_D$ or $\overrightarrow{R}^B_{\overline{D}}$, and $\overrightarrow{R}^A_{\overline{D}}$or $\overrightarrow{R}^B_D$ in order to appear on $\overrightarrow{R}^C_0$. Finally, 0 is eliminated from the output if C is assigned to a value other than 0 during the test generation process. The complete set of reduction equations for output C of a two-input *AND* gate is:

$$\begin{aligned}
\overrightarrow{R}^C_0 &= (\overrightarrow{R}^A_0 \cap \overrightarrow{R}^B_0 \cap (\overrightarrow{R}^A_{\overline{D}} \cup \overrightarrow{R}^B_D) \cap (\overrightarrow{R}^A_D \cup \overrightarrow{R}^B_{\overline{D}})) \\
&\quad \cup \{C_{\{\overline{D},D,1\}}\} \\
\overrightarrow{R}^C_{\overline{D}} &= ((\overrightarrow{R}^A_1 \cup \overrightarrow{R}^B_{\overline{D}}) \cap (\overrightarrow{R}^A_{\overline{D}} \cup \overrightarrow{R}^B_1) \cap (\overrightarrow{R}^A_{\overline{D}} \cup \overrightarrow{R}^B_{\overline{D}})) \\
&\quad \cup \{C_{\{0,D,1\}}\} \\
\overrightarrow{R}^C_D &= ((\overrightarrow{R}^A_1 \cup \overrightarrow{R}^B_D) \cap (\overrightarrow{R}^A_D \cup \overrightarrow{R}^B_1) \cap (\overrightarrow{R}^A_D \cup \overrightarrow{R}^B_D)) \\
&\quad \cup \{C_{\{0,\overline{D},1\}}\} \\
\overrightarrow{R}^C_1 &= \overrightarrow{R}^A_1 \cup \overrightarrow{R}^B_1 \cup \{C_{\{0,\overline{D},D\}}\}.
\end{aligned}$$

Similarly, the reduction equations for output C of a two-input *XOR* gate are:

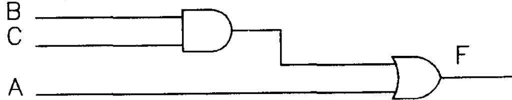
$$\begin{aligned}
\overrightarrow{R}^C_0 &= ((\overrightarrow{R}^A_0 \cup \overrightarrow{R}^B_0) \cap (\overrightarrow{R}^A_{\overline{D}} \cup \overrightarrow{R}^B_{\overline{D}}) \cap (\overrightarrow{R}^A_D \cup \overrightarrow{R}^B_D) \cap \\
&\quad (\overrightarrow{R}^A_1 \cup \overrightarrow{R}^B_1)) \cup \{C_{\{\overline{D},D,1\}}\} \\
\overrightarrow{R}^C_{\overline{D}} &= ((\overrightarrow{R}^A_0 \cup \overrightarrow{R}^B_{\overline{D}}) \cap (\overrightarrow{R}^A_{\overline{D}} \cup \overrightarrow{R}^B_0) \cap (\overrightarrow{R}^A_D \cup \overrightarrow{R}^B_1) \cap \\
&\quad (\overrightarrow{R}^A_1 \cup \overrightarrow{R}^B_D)) \cup \{C_{\{0,D,1\}}\} \\
\overrightarrow{R}^C_D &= ((\overrightarrow{R}^A_0 \cup \overrightarrow{R}^B_D) \cap (\overrightarrow{R}^A_{\overline{D}} \cup \overrightarrow{R}^B_1) \cap (\overrightarrow{R}^A_D \cup \overrightarrow{R}^B_0) \cap \\
&\quad (\overrightarrow{R}^A_1 \cup \overrightarrow{R}^B_{\overline{D}})) \cup \{C_{\{0,\overline{D},1\}}\} \\
\overrightarrow{R}^C_1 &= ((\overrightarrow{R}^A_0 \cup \overrightarrow{R}^B_1) \cap (\overrightarrow{R}^A_{\overline{D}} \cup \overrightarrow{R}^B_D) \cap (\overrightarrow{R}^A_D \cup \overrightarrow{R}^B_{\overline{D}}) \cap \\
&\quad (\overrightarrow{R}^A_1 \cup \overrightarrow{R}^B_0)) \cup \{C_{\{0,\overline{D},D\}}\}.
\end{aligned}$$

Example 4: The circuit in Fig. 7, taken from [14], illustrates the concept of reduction lists. Here, we see that $A_{\{1\}}$ appears on $\overrightarrow{R}^F_0$. Thus, during test pattern generation, if $F_{\{0\}}$ is required, then $\{1\}$ must be eliminated at stem A. Note that the circuit from Fig. 7 is nonminimal, implementing the same function as the circuit shown in Fig. 8b[1].

[1] In fact, it is a general property of the reduction equations that if the reduction lists of a reconvergence gate are not empty when all inputs are assigned to $\{0,1\}$, then the circuit in question is not a minimal representation, and can be redesigned to become both smaller and easier to test.

Line	List	Contents
A	$\overrightarrow{R}_0^A$ $\overrightarrow{R}_1^A$	$\{A_{\{1\}}\}$ $\{A_{\{0\}}\}$
B	$\overrightarrow{R}_0^B$ $\overrightarrow{R}_1^B$	$\{B_{\{1\}}\}$ $\{B_{\{0\}}\}$
C	$\overrightarrow{R}_0^C$ $\overrightarrow{R}_1^C$	$\{C_{\{1\}}\}$ $\{C_{\{0\}}\}$
D	$\overrightarrow{R}_0^D$ $\overrightarrow{R}_1^D$	$\{A_{\{1\}}, B_{\{1\}}\}$ $\{\}$
E	$\overrightarrow{R}_0^E$ $\overrightarrow{R}_1^E$	$\{A_{\{1\}}, C_{\{1\}}\}$ $\{\}$
F	$\overrightarrow{R}_0^F$ $\overrightarrow{R}_1^F$	$\{A_{\{1\}}\}$ $\{\}$

a) Reduction lists for the circuit from Fig. 7



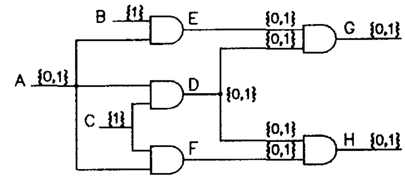
b) Minimized example circuit

Figure 8 Necessary assignments in a circuit

Example 5: The circuit in Fig. 9 illustrates the use of reduction lists when circuit values are partially determined. Here, we see that $A_{\{1\}}$ reduces $G_{\{0\}}$ and $H_{\{0\}}$ if $B = \{1\}$ and $C = \{1\}$ have already been determined by other assignments during test pattern generation. However, if $A = \{0,1\}$, $B = \{0,1\}$, and $C = \{0,1\}$, then $A_{\{1\}}$ will not reduce either $G_{\{0\}}$ or $H_{\{0\}}$.

4.2 Logical constraints and propagation of implications

By formulating the test pattern generation problem in terms of images and inverse images of set functions, rather than in terms of logical assignments and their implications, we are able to extract information about the function of the circuit under test. This is important, as we are interested in the logical constraints imposed by assignments, rather than in the signal values these assignments produce. Logical constraints, unlike signal values, propagate in both directions in the circuit—from inputs toward outputs and from outputs toward inputs. The result of full implication propagation is to determine *all* implications of each assignment, both forward and backward in the circuit.



a) Example circuit

Line	List	Contents
A	$\overrightarrow{R}_0^A$ $\overrightarrow{R}_1^A$	$\{A_{\{1\}}\}$ $\{A_{\{0\}}\}$
B	$\overrightarrow{R}_1^B$	$\{\}$
C	$\overrightarrow{R}_1^C$	$\{\}$
D	$\overrightarrow{R}_0^D$ $\overrightarrow{R}_1^D$	$\{A_{\{1\}}, D_{\{1\}}\}$ $\{A_{\{0\}}, D_{\{0\}}\}$
E	$\overrightarrow{R}_0^E$ $\overrightarrow{R}_1^E$	$\{A_{\{1\}}\}$ $\{A_{\{0\}}\}$
F	$\overrightarrow{R}_0^F$ $\overrightarrow{R}_1^F$	$\{A_{\{1\}}\}$ $\{A_{\{0\}}\}$
G	$\overrightarrow{R}_0^G$ $\overrightarrow{R}_1^G$	$\{A_{\{1\}}\}$ $\{A_{\{0\}}, D_{\{0\}}\}$
H	$\overrightarrow{R}_0^H$ $\overrightarrow{R}_1^H$	$\{A_{\{1\}}\}$ $\{A_{\{0\}}, D_{\{0\}}\}$

b) Reduction lists

Figure 9 Necessary assignments with partially determined circuit values

Test pattern generation algorithms which focus on the logical effect of assignments in the circuit fail to determine the effect of logic dependencies between nodes in the circuit. Traversal of a circuit module does not guarantee that sufficient information will be extracted about its function to enable the algorithm to reason about the module. Some algorithms resort to defining common modules as logic blocks recognized by the test pattern generation program so that the function of the module is known without traversing the structure [13]. Our algorithm is able to extract this information automatically.

Example 6: In order to test the fault F/s_0 in the subcircuit shown in Fig. 10, we must justify the assignment $F_{\{1\}}$. We note that $E_{\{0\}}$ requires $A_{\{0\}}$, as $A_{\{1\}}$ reduces $E_{\{0\}}$. Since $E_{\{0\}}$ appears on $\overleftarrow{R}_1^{a2}$, it also appears on $\overrightarrow{R}_1^{a1}$, and thus

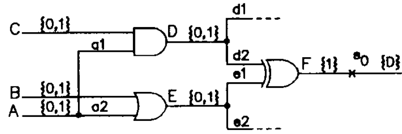
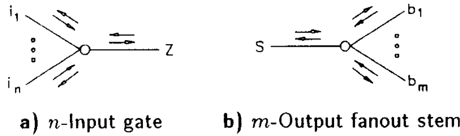


Figure 10 Test generation for fault F/s_0

propagates to stem D and appears on $\overrightarrow{R}^{D}_{1}$. Since $E_{\{0\}}$ appears on both $\overrightarrow{R}^{d_2}_{1}$ and $\overrightarrow{R}^{e1}_{1}$, it reduces $F_{\{1\}}$. However, $F_{\{1\}}$ is a justification point—thus $E_{\{0\}}$ is a reduction assignment, and $E_{\{1\}}$ is necessary. A similar argument applies to $D_{\{0\}}$, which is also necessary. It is important to observe that an assignment to stem E appears on a reduction list at stem D despite the fact that D is neither driven by nor drives E—the constraint has traveled to a region of the circuit to which a logic value cannot.



a) n-Input gate **b)** m-Output fanout stem

Figure 11 Circuit nodes with associated reduction lists

The circuit under test can be viewed as a graph, with gates represented by nodes and lines as edges. Primary inputs and outputs are special types of gates, with no inputs and no outputs, respectively. Each edge in the graph has a set of reduction lists associated with it, one reduction list for each possible value of the line; each reduction list is composed of two components which are distinguished as "forward" ($\overrightarrow{\ }$) and "backward" ($\overleftarrow{\ }$). Circuit nodes relate the reduction lists of the edges connected to them, as shown in Fig. 11. For each edge connected to the node, its outward pointing reduction list (forward or backward if the line is an output from or input to the corresponding gate, respectively) is a function of the inward pointing reduction lists of all other edges connected to the node—the type of gate determines the function performed on the lists.

In order to reduce a value from an input of a gate, it is necessary to eliminate all combinations of values of the output and other input(s) of the gate which use the reduced input value. For example, from the inverse-image characteristic equations for a two-input *AND* gate given in section 3, value 1 remains in the set of possible values at input A of the gate if 0, 1, D, or $\overline{D}$ is present at input B and remains in the implied value of the output. Thus, in order for an assignment to reduce 1 at input A, it must reduce: either 0 at input B or 0 at output C, either 1 at input A or 1 at output C, either D at input B or D at output C, and either $\overline{D}$ at input B or $\overline{D}$ at output C. In other words, an assignment must appear on $\overrightarrow{R}^{B}_{0}$ or $\overleftarrow{R}^{C}_{0}$, $\overrightarrow{R}^{B}_{1}$ or $\overleftarrow{R}^{C}_{1}$, $\overrightarrow{R}^{A}_{D}$ or $\overleftarrow{R}^{C}_{D}$, and $\overrightarrow{R}^{B}_{\overline{D}}$ or $\overleftarrow{R}^{C}_{\overline{D}}$ in order to appear on $\overleftarrow{R}^{A}_{1}$. The complete set of reduction equations for input A of a two-input *AND* gate is:

$$\overleftarrow{R}^{A}_{0} = \overleftarrow{R}^{C}_{0} \cup \{A_{\{1,\overline{D},D\}}\}.$$

$$\overleftarrow{R}^{A}_{\overline{D}} = ((\overrightarrow{R}^{B}_{1} \cup \overleftarrow{R}^{C}_{\overline{D}}) \cap (\overrightarrow{R}^{B}_{D} \cup \overleftarrow{R}^{C}_{0}) \cap (\overrightarrow{R}^{B}_{\overline{D}} \cup \overleftarrow{R}^{C}_{\overline{D}}) \cap (\overrightarrow{R}^{B}_{0} \cup \overleftarrow{R}^{C}_{0})) \cup \{A_{\{0,D,1\}}\}$$

$$\overleftarrow{R}^{A}_{D} = ((\overrightarrow{R}^{B}_{1} \cup \overleftarrow{R}^{C}_{D}) \cap (\overrightarrow{R}^{B}_{D} \cup \overleftarrow{R}^{C}_{D}) \cap (\overrightarrow{R}^{B}_{\overline{D}} \cup \overleftarrow{R}^{C}_{0}) \cap (\overrightarrow{R}^{B}_{0} \cup \overleftarrow{R}^{C}_{0})) \cup \{A_{\{0,\overline{D},1\}}\}$$

$$\overleftarrow{R}^{A}_{1} = ((\overrightarrow{R}^{B}_{1} \cup \overleftarrow{R}^{C}_{1}) \cap (\overrightarrow{R}^{B}_{D} \cup \overleftarrow{R}^{C}_{D}) \cap (\overrightarrow{R}^{B}_{\overline{D}} \cup \overleftarrow{R}^{C}_{\overline{D}}) \cap (\overrightarrow{R}^{B}_{0} \cup \overleftarrow{R}^{C}_{0})) \cup \{A_{\{0,\overline{D},D\}}\}.$$

Similarly, for input A of a two-input *XOR* gate, the reduction equations are:

$$\overleftarrow{R}^{A}_{0} = ((\overleftarrow{R}^{C}_{0} \cup \overrightarrow{R}^{B}_{0}) \cap (\overleftarrow{R}^{C}_{\overline{D}} \cup \overrightarrow{R}^{B}_{\overline{D}}) \cap (\overleftarrow{R}^{C}_{D} \cup \overrightarrow{R}^{B}_{D}) \cap (\overleftarrow{R}^{C}_{1} \cup \overrightarrow{R}^{B}_{1})) \cup \{A_{\{\overline{D},D,1\}}\}$$

$$\overleftarrow{R}^{A}_{\overline{D}} = ((\overleftarrow{R}^{C}_{0} \cup \overrightarrow{R}^{B}_{\overline{D}}) \cap (\overleftarrow{R}^{C}_{\overline{D}} \cup \overrightarrow{R}^{B}_{0}) \cap (\overleftarrow{R}^{C}_{D} \cup \overrightarrow{R}^{B}_{1}) \cap (\overleftarrow{R}^{C}_{1} \cup \overrightarrow{R}^{B}_{D})) \cup \{C_{\{0,D,1\}}\}$$

$$\overleftarrow{R}^{A}_{D} = ((\overleftarrow{R}^{C}_{0} \cup \overrightarrow{R}^{B}_{D}) \cap (\overleftarrow{R}^{C}_{\overline{D}} \cup \overrightarrow{R}^{B}_{1}) \cap (\overleftarrow{R}^{C}_{D} \cup \overrightarrow{R}^{B}_{0}) \cap (\overleftarrow{R}^{C}_{1} \cup \overrightarrow{R}^{B}_{\overline{D}})) \cup \{C_{\{0,\overline{D},1\}}\}$$

$$\overleftarrow{R}^{A}_{1} = ((\overleftarrow{R}^{C}_{0} \cup \overrightarrow{R}^{B}_{1}) \cap (\overleftarrow{R}^{C}_{\overline{D}} \cup \overrightarrow{R}^{B}_{D}) \cap (\overleftarrow{R}^{C}_{D} \cup \overrightarrow{R}^{B}_{\overline{D}}) \cap (\overleftarrow{R}^{C}_{1} \cup \overrightarrow{R}^{B}_{0})) \cup \{C_{\{0,\overline{D},D\}}\}.$$

For each fanout branch of the fanout stem shown if Fig. 11b,

$$\overrightarrow{R}^{b_i}_{v} = \overrightarrow{R}^{S}_{v} \cup \bigcup_{j \neq i} \overleftarrow{R}^{b_j}_{v} \qquad \text{for each } v \in S.$$

For the fanout stem itself,

$$\overleftarrow{R}^{S}_{v} = \bigcup_{j=1}^{n} \overleftarrow{R}^{b_j}_{v} \qquad \text{for each } v \in S.$$

Dependencies between circuit nodes can cause logical constraints at one justification point to appear at another. The reduction equations are able to capture these constraints and determine additional necessary assignments.

Example 7: In order to produce a "1" on both the sum and carry outputs of the full adder from Fig. 12, we need a "1" on both the A, B, and C (carry-in) inputs. As noted in [13], there are no direct implications of an assignment to A, B, or C. However, assignment $C_{\{0\}}$ appears on $\overrightarrow{R}^{F}_{1}$: since $H_{\{1\}}$ must be justified, an assignment which reduces $F_{\{1\}}$ implies that E must carry value "1" (*i.e.* $C_{\{0\}}$ appears on $\overleftarrow{R}^{E}_{0}$). Thus, assignment $C_{\{0\}}$ requires that $A = B = 1$ (*i.e.*

appears on both $\overleftarrow{R}_0^{a_2}$ and $\overleftarrow{R}_0^{b_2}$), since "1" must appear on line E if $C = 0$ in order to satisfy requirement $H_{\{1\}}$. Thus, $C_{\{0\}}$ appears on $\overrightarrow{R}_1^{D}$, since it appears on both $\overrightarrow{R}_0^{a_1}$ and $\overrightarrow{R}_0^{b_1}$. Finally, since $C_{\{0\}}$ appears on $\overrightarrow{R}_1^{D}$ and $\overrightarrow{R}_1^{C}$, it appears on $\overrightarrow{R}_1^{G}$. Since $G_{\{1\}}$ is a justification point, $C_{\{0\}}$ is a reduction assignment and $C_{\{1\}}$ is necessary. Despite that there are no local implications from assignment $C_{\{0\}}$, it appears on $\overrightarrow{R}_1^{G}$. A similar argument applies to both assignment $A_{\{1\}}$ and $B_{\{1\}}$, both of which are also necessary.

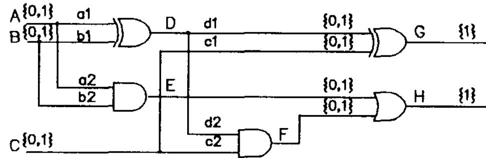


Figure 12 Value justification of a full adder

4.3 Stem Correlation

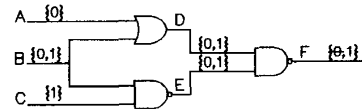
If the values at the inputs to a gate are not independent—that is, they are related to one another through reconvergence of some fanout stem—then the set of possible values at the output of the gate may be pessimistic [14]. The reduction lists allow us to eliminate this pessimism.

Example 8: The circuit of Fig. 13 is an implementation of a two-input multiplexor (*MUX*) with select B and inputs $\overline{A}$, C. Since $\overrightarrow{R}_0^{F} = \{B_{\{0,1\}}\}$, *any* of the possible assignments to stem B cause 0 to vanish at line F. A *stem correlation* exists for assignment $F_{\{0\}}$ caused by B. Therefore, 0 is not an attainable assignment at line F and the forward propagated value is $\{1\}$ rather than $\{0,1\}$. Stem correlation turns out to be extremely important in test pattern generation, reducing the number of branches and backtracks as well as the total CPU time required to generate a test.

4.4 Implementation issues

The complexity of reduction list calculation is similar to that of deductive fault simulation. During the test generation process, reduction lists can only grow—assignments can only be added, never deleted[2]. Since the total number of assignments which can appear on any reduction list cannot be greater than the number of nodes in the circuit, the reduction list calculation will complete in polynomial time.

The amount of processing which is required to determine the reduction lists is directly proportional to the number of assignments which must be analysed and the area of the



a) 2-input *MUX*

Line	List	Contents
A	$\overrightarrow{R}_0^{A}$	$\{\}$
B	$\overrightarrow{R}_0^{B}$	$\{B_{\{1\}}\}$
	$\overrightarrow{R}_1^{B}$	$\{B_{\{0\}}\}$
C	$\overrightarrow{R}_1^{C}$	$\{\}$
D	$\overrightarrow{R}_0^{D}$	$\{B_{\{1\}}\}$
	$\overrightarrow{R}_1^{D}$	$\{B_{\{0\}}\}$
E	$\overrightarrow{R}_0^{E}$	$\{B_{\{0\}}\}$
	$\overrightarrow{R}_1^{E}$	$\{B_{\{1\}}\}$
F	$\overrightarrow{R}_0^{F}$	$\{B_{\{0,1\}}\}$
	$\overrightarrow{R}_1^{F}$	$\{\}$

b) Reduction lists

Figure 13 Correlation of assignments

circuit in which they must be propagated. The concepts of *stem regions* and *exit lines* [10] can be used to restrict both while guaranteeing that no necessary assignments will be overlooked [6].

Since the dynamic calculation of reduction lists may be costly, it may be desirable to obtain generic information which can be reused for each target fault. Analogous to "static learning" [14], the "initial" reduction lists (when all node assignments are $\{0,1\}$ and no fault has been injected) are preprocessed and retained. These lists contain reduction assignments which can be used for justification points which are not reachable from the point of the fault.

It is not necessary to explicitly extract *dominators*—nodes through which the fault effect must propagate in order to reach the primary outputs—if the reduction equations are updated dynamically. Dominators are simply necessary assignments to $\{D, \overline{D}\}$ rather than to $\{0\}$ or $\{1\}$. If the reduction lists are not updated dynamically, then dominators can be extracted by tracing backward from those justification points whose required value includes D and/or $\overline{D}$ using a simple linear algorithm [5]. In this case, another preprocessing step is performed to find those assignments which are necessary to propagate a fault effect from a stem to its *exit lines*. These are reduction assignments which must be eliminated whenever the stem is a dominator [6]. Preprocessed "propagate assignments" turn out to be extremely useful for certain difficult faults.

Experimental results show that dynamic calculation of re-

[2] The lists may decrease in size on a backtrack, as we move from a later stage of test generation to an earlier one where the lists were shorter.

duction lists is not required to generate a test for or prove redundant the vast majority of faults.

5. Experimental results

The algorithms presented in this paper have been implemented in C running under UNIX. The goal of the implementation was to investigate the behavior of an algorithm which uses necessary assignments[3] rather than "intelligent" heuristics for test pattern generation and, in particular, to determine the extent to which backtracking can be reduced.

In order to test the deductive power of the algorithm, all faults were explicitly targeted. In order to avoid the fortuitous detection of a difficult fault with a lucky random test pattern, no fault simulation was performed. The choice of heuristic for choosing arbitrary branches has a huge impact on the number of backtracks performed and the number of abandoned faults. Several different heuristics for choosing arbitrary branches were implemented, but none were found which worked well in all circuits. The results presented in this section were produced by assigning to {0} the first unassigned primary input which could have an effect in the final test pattern whenever no necessary or nonconflicting assignments could be found.

Complete test pattern generation experiments were run on the ISCAS'85 benchmark circuits [2]. The results were generated as follows:

1. *Deterministic test pattern generation:* All faults were explicitly targeted. A two-pass algorithm was used, with a backtrack limit of 10 for each pass:
   a) *Phase 1:* Test generation using preprocessed reduction lists only, but using dominators and necessary assignments for propagation from stems to their exit line(s).
   b) *Phase 2:* Test generation using dynamically calculated reduction lists[4]
2. *Fault simulation:* The test set obtained in phase 1 and phase 2 was fault simulated [10] to determine the coverage of abandoned faults. "Don't cares" in the test patterns were randomly replaced with zeros and ones. Other than replacement of "don't care" values, no random pattern fault simulation was performed.

[3] In order to achieve these results, algorithms which determine nonconflicting assignments have also been used [6].

[4] The current implementation does not take advantage of the stem region concept.

Circuit	No. Flts.	Abd. Flts.	total bktrks.	CPU Time (s)*		
				Avg.	Max.	Dev.
C432	4	0	31	1.51	2.01	0.48
C499	8	0	0	0.40	0.66	0.21
C1355	8	0	0	0.66	0.76	0.04
C1908	9	0	0	0.41	0.54	0.09
C2670	117	0	8	0.89	4.52	0.94
C3540	137	0	0	1.67	6.38	1.10
C5315	59	0	0	0.35	2.24	0.38
C6288	34	0	0	0.42	0.56	0.13
C7552	131	0	14	0.89	3.38	0.50

* Sun 3/60

Table 3 Experimental results—redundant faults

Table 3 summarizes the results for the untestable faults in the benchmark circuits. For each circuit, the table gives the number of redundant faults, the number of abandoned faults (faults which were not proven to be redundant), the total number of backtracks performed in the experiment, and the average, maximum and standard deviation of the times required to prove the faults redundant.

No redundant faults were abandoned. Furthermore, phase 2 of the test pattern generation algorithm (full reduction list propagation) was required only for circuit C432—three faults were abandoned (after 10 backtracks each) by phase 1, after which two of the three were proven redundant with no backtracks and the third was proven redundant with one backtrack in phase 2. Circuit C2670 contains eight faults which each required one backtrack (in phase 1) to prove redundancy; circuit C7552 contains 6 faults which were backtracked, the "worst" of which required 3 backtracks (phase 1).

Table 4 gives the results for the testable faults of the benchmark circuits. For each circuit, the table gives the num-

Circuit	No. Flts.	Faults tested:		Abd. faults:		CPU Time (s)			
		w/o Brnch.	w/o Bcktrk.	w/o FS	w FS	Avg.	Max.	Dev.	Pre.
C432	540	189	538	0	0	0.80	1.80	0.25	3.76
C499	750	64	719	0	0	1.11	3.72	0.61	8.84
C880	942	402	940	0	0	0.56	2.02	0.49	8.84
C1355	1566	0	1480	0	0	3.55	6.36	1.59	35.36
C1908	1870	210	1840	3	0	2.94	7.52	1.53	39.26
C2670	2630	553	2490	20	0	1.57	6.96	1.00	54.68
C3540	3291	611	2654	49	0	3.39	11.40	1.89	159.22
C5315	5291	1242	5122	30	0	1.13	6.28	0.99	102.74
C6288	7710	12	6197	1127	0	7.87	17.74	2.67	210.70
C7552	7419	203	6348	39	0	2.52	11.74	1.51	148.06

Table 4 Experimental results—testable faults

ber of testable faults, the number of faults for which a test pattern was generated without any arbitrary branching and without any backtracking, the number of testable faults for which the algorithm was unable to find a test vector, the number of faults which remained undetected after vectors generated for the other faults were fault simulated, the average time, maximum time and standard deviation of times required to generate a test for each target, and the time required to preprocess the circuit.

Of particular interest is the number of faults for which a test is generated with no arbitrary branching. An interesting result is that several faults exist which were abandoned by phase 1 of the test pattern generation algorithm, but for which a test pattern was generated without branching by phase 2. Another interesting result is that all of the faults which were abandoned after phases 1 and 2 of the test pattern generation algorithm were covered when less than 1000 random vectors were fault simulated, and thus would not have been targeted in a conventional test pattern generation experiment.

6. Conclusions

We have presented a new test pattern generation algorithm which uses the concept of *necessary* assignments to reduce or eliminate backtracking in automatic test pattern generation. Necessary assignments are those which must be made in order to find a test pattern; the search is guaranteed to fail if we do not make them.

This concept has been incorporated into an automatic test pattern generation algorithm which has been used to generate test patterns for all faults in a variety of benchmark circuits. Experimental results indicate that the algorithm is particularly efficient at *redundancy identification* which is often a problem for conventional test pattern generation algorithms.

It is interesting to compare the problems of test pattern generation and fault simulation. The local implication step used by PODEM and FAN for test pattern generation can be compared to critical path tracing in fault simulation. They are of similar computational complexity; critical path tracing is exact and conventional implication will lead to a test pattern with no branching only in fanout-free circuits. Finding the implication of each possible assignment in SOCRATES is similar to serial fault simulation, whereas the processing of reduction lists is similar to deductive fault simulation. In this light, we can see the progression from heuristic to algorithmic test pattern generation algorithms.

References

[1] S.B. Akers, "A Logic System for Fault Test Generation," *IEEE Transactions on Computers*, vol. C-25, no. 2, June, 1976, pp. 620–630.

[2] F. Brglez, and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," *Proceedings International Symposium on Circuits and Systems, Special Sesson on ATPG and Fault Simulation*, Kyoto, Japan, June, 1985.

[3] C. Cha, W. Donath, and F. Özgüner, "9-V Algorithm for Test Pattern Generation of Combinational Digital Circuits," *IEEE Transactions on Computers*, vol. C-27, no. 3, March 1978, pp. 193–200.

[4] W.T. Cheng, "Split Circuit Model for Test Generation," *Proceedings 25th Design Automation Conference*, Anaheim, CA, June, 1988, pp. 96–101.

[5] H. Cox, and J. Rajski, "A Method of Test Generation and Fault Diagnosis," *IEEE Transactions on Computer-Aided Design*, vol. 7, no. 7, July, 1988, pp. 813–833.

[6] H. Cox, *Properties of Set Functions and Their Application to Test Pattern Generation*, Ph.D. Thesis, Department of Electrical Engineering, McGill University, September 1990.

[7] H. Fujiwara, and T. Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Transactions on Computers*, vol. C-32, no. 12, December, 1983, pp. 1137–1144.

[8] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Transactions on Computers*, vol. C-30, no. 3, March, 1981, pp. 215–222.

[9] T. Kirkland, and M.R. Mercer, "A Topological Search Algorithm for ATPG," *Proceedings 24th Design Automation Conference*, Miami Beach, FL, June, 1987, pp. 502–508.

[10] F. Maamari, and J. Rajski, "A Method of Fault Simulation Based on Stem Regions," *IEEE Transactions on Computer-Aided Design*, vol. 9, no. 2, February 1990, pp. 212–220.

[11] P. Muth, "A Nine-Valued Circuit Model for Test Generation," *IEEE Transactions on Computers*, vol. C-25, no. 6, June 1976, pp. 630–636.

[12] J. Rajski, "GEMINI: A Logic System for Fault Diagnosis Based on Set Functions," *Digest 18th International Symposium on Fault-Tolerant Computing Systems*, Tokyo, Japan, June, 1988, pp. 292–297.

[13] T.M. Sarfert, R. Markgraf, E. Trischler, and M.H. Schulz, "Hierarchical Test Pattern Generation Based on High-Level Primitives," *Proceedings International Test Conference*, Washington DC, August 1989, pp. 470–479.

[14] M.H. Schulz, and E. Auth, "Improved Deterministic Test Pattern Generation With Applications to Redundancy Identification," *IEEE Transactions on CAD*, vol. 8, no. 7, July 1989, pp. 811–816.