Resynthesis of Combinational Logic Circuits for Improved Path Delay Fault Testability Using Comparison Units

Irith Pomeranz, Fellow, IEEE, and Sudhakar M. Reddy, Fellow, IEEE

Abstract—We propose a resynthesis method that modifies a given circuit to reduce the number of paths in the circuit and thus improve its path delay fault testability. The resynthesis procedure is based on replacing subcircuits of the given circuit by structures called comparison units. A subcircuit can be replaced by a comparison unit if it implements a function belonging to the class of comparison functions defined here. Comparison units are fully testable for stuck-at faults and for path delay faults. In addition, they have small numbers of paths and gates. These properties make them effective building blocks for resynthesis to improve the path delay fault testability of a circuit. Experimental results demonstrate considerable reductions in the number of paths and increased path delay fault testability. These are achieved without increasing the number of gates, or the number of gates along the longest path in the circuit. The random pattern testability for stuck-at faults remains unchanged.

Index Terms—Combinational circuits, design-for-testability, path delay faults.

I. INTRODUCTION

In this paper, we define a special class of functions, called comparison functions. Informally stated, a comparison function can be specified by providing a permutation \( \mathbf{X} \) of its input variables and two bounds \( \mathbf{L} \) and \( \mathbf{U} \). Under the input permutation \( \mathbf{X} \), every minterm where the function assumes a value of one has a decimal value between \( \mathbf{L} \) and \( \mathbf{U} \). Comparison functions have the property that they can be implemented by circuits referred to here as comparison units. Comparison units are efficient in terms of the number of gates they require, they have a small number of paths going through them, and all the path delay faults in them are robustly testable. Consequently, comparison functions are useful in synthesizing circuits with improved path delay fault testability.

Comparison functions are utilized in this paper to reduce the path count of combinational circuits through resynthesis based on local circuit modifications. During the resynthesis process, subcircuits realizing comparison functions are identified and replaced by comparison units whenever such a replacement reduces the number of paths or the number of gates in the circuit. Area reduction by resynthesis based on local circuit modifications was considered in [1]–[4]. Local modifications to enhance testability for path delay faults were considered in [5]–[7]. The advantage of comparison functions in guiding local circuit modifications is that comparison functions provide a simple and uniform method of selecting the subcircuits that will be modified and their new structure. In addition, the use of comparison functions results in significant reductions in the number of paths. These reductions are larger than any reductions reported previously. Moreover, experimental results demonstrate that the reduction in the number of paths results mainly in reducing the number of untestable path delay faults, while the number of testable paths increases. Thus, the path delay fault testability is enhanced significantly.

Next, we consider in more detail the motivation for reducing the number of paths in a circuit. The path delay fault model was proposed to model defects that change the timing behavior of a circuit [8]. It is the most general of all delay fault models, since it models distributed as well as localized excessive delays. However, three problems are associated with this fault model that prevent test generation procedures from achieving complete or close-to-complete fault coverage.

1) The number of paths (and therefore the number of path delay faults) in practical circuits may be very large [9].
2) The number of tests to detect all the path delay faults may be very large [6].
3) Many path delay faults in practical circuits are not testable [10].

The problem of handling large numbers of paths was alleviated in part by the nonenumerative methods of [9] and [11]. However, even using these techniques, the fault coverage for large circuits is very low. This is due in part to the large number of tests required to detect all the faults, and in part to the fact that many of the faults are untestable. In [12] and [13], it was shown that some path delay faults do not have to be tested, as correct speed of operation can be guaranteed by testing other faults. However, even when using this approach, the fault coverage obtained is sometimes low. In [14], a test-point insertion method to increase the testability of a circuit to path delay faults was presented. However, test-point insertion has the disadvantages of area and test application overheads, especially if a large number of test points are needed to achieve the desired fault coverage. All the methods above, as well as test generation and fault simulation methods, can benefit from a reduction in the number of paths in the circuit under consideration. This is achieved by...

Manuscript received March 8, 2000. This work was supported in part by the National Science Foundation under Grants MIP-9220549 and MIP-9357581. This paper is based on “On Synthesis-for-Testability of Combinational Logic Circuits,” Proceedings of ACM/IEEE 32nd Design Automation Conference, June 1995, pp. 126–132.

I. Pomeranz is with the School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907 USA.

S. M. Reddy is with the Electrical and Computer Engineering Department, University of Iowa, Iowa City, IA 52242 USA.
the proposed resynthesis procedure that uses comparison functions.

This paper is organized as follows. The required background is presented in Section II. Comparison functions, circuits to implement them, and their relationship to threshold functions are described in Section III. In Section IV, we describe resynthesis procedures based on comparison functions. Experimental results are presented in Section V. Section VI describes generalized comparison functions, which are an extension of comparison functions. Section VII concludes this paper.

II. PRELIMINARIES

To compute the number of paths in a given circuit, we use the following procedure from [9]. The procedure attaches a label \( N_p(g) \) to each line \( g \), equal to the number of paths from the primary inputs to line \( g \). The procedure starts by assigning to every primary input the label 1. It then proceeds from inputs to outputs. The output of a gate is labeled by the sum of its input labels. A fanout branch is labeled by the same label assigned to its stem. The total number of paths is equal to the sum of the primary output labels.

Our goal in this paper is to reduce the number of paths in a circuit by performing local modifications to the circuit. To demonstrate the effect that such modifications can have on the number of paths, consider a \( k \)-input single-output subcircuit \( C' \) with inputs \( \{g_i; 1 \leq i \leq k\} \) and output \( g \), as shown in Fig. 1. Suppose that inside \( C' \), there are \( K_p(g_i, g) \) paths from \( g_i \) to \( g \). The number of paths from the primary inputs to \( g \) can be expressed as \( N_p(g) = \sum_{i=1}^{k} [N_p(g_i) \cdot K_p(g_i, g)] \) where \( N_p(g_i) \) for the number of paths from the primary inputs to \( g_i \) and \( K_p(g_i, g) \) for the number of paths from \( g_i \) to \( g \). If we change the structure of \( C' \) such that smaller values of \( K_p(g_i, g) \) will be obtained, or such that smaller values of \( K_p(g_i, g) \) will correspond to the larger values of \( N_p(g_i) \), we will reduce the number of paths to \( g \) and, consequently, the total number of paths in the circuit. The following example demonstrates the case where smaller values of \( K_p(g_i, g) \) are matched with larger values of \( N_p(g_i) \) in order to reduce the total number of paths.

Example: Consider the four-input single-output function \( f_1 \) that has the two (equivalent) minimal sum-of-products expressions \( f_{1,1} = \overline{x_1}x_2x_3 + x_1\overline{x_2}x_3 + x_2\overline{x_3}x_4 \) and \( f_{1,2} = \overline{x_1}x_2x_4 + x_1\overline{x_2}x_3 + x_2\overline{x_3}x_4 \). Assuming a two-level implementation, \( K_p(x_i, f_1) \) is equal to the number of times \( x_i \) or \( \overline{x_i} \) appears in the expression for \( f_1 \). Thus, for the first implementation \( (f_{1,1}) \), we have \( K_{p,1}(x_1; f_{1,1}) = 2, K_{p,1}(x_2; f_{1,1}) = 3, K_{p,1}(x_3; f_{1,1}) = 2, \) and \( K_{p,1}(x_4; f_{1,1}) = 2 \). For the second implementation \( (f_{1,2}) \), we have \( K_{p,2}(x_1; f_{1,2}) = 3, K_{p,2}(x_2; f_{1,2}) = 2, K_{p,2}(x_3; f_{1,2}) = 2, \) and \( K_{p,2}(x_4; f_{1,2}) = 2 \). Suppose that \( f_1 \) is implemented by a subcircuit \( C' \) embedded in a circuit \( C \). Let \( N_p(x_1) = 10, N_p(x_2) = 100, N_p(x_3) = 20, \) and \( N_p(x_4) = 20 \), as shown in Fig. 2 (numbers of paths are given on lines connecting the circuit inputs to the inputs of \( C' \), and on lines connecting the inputs and the output of \( C' \); e.g., the line with \( N_p = 10 \) indicates that ten paths exist from the inputs of \( C \) to \( x_1 \). Under the first implementation, \( N_p(f_{1,1}) = 2 \cdot 10 + 3 \cdot 100 + 2 \cdot 20 + 2 \cdot 20 = 310 \). This is a result of the fact that \( K_{p,1}(x_1; f_{1,1}, f_{1,1}) < K_{p,2}(x_1, f_{1,2}) \), \( K_{p,1}(x_2, f_{1,1}) > K_{p,2}(x_2, f_{1,2}), \) and \( N_p(x_1) < N_p(x_2) \).

The problem of reducing the number of paths by local circuit modifications is the following. Given a circuit \( C \), find a set of subcircuits \( C_1', C_2', ..., C_k' \) of \( C \) and a set of subcircuits \( D_1', D_2', ..., D_k' \), such that if \( C_i' \) is replaced by \( D_i' \), 1 \( \leq i \leq k \), the function implemented by the circuit \( C \) does not change, and the number of paths in \( C \) is minimum.

To simplify the problem, we impose the following constraints.

1) We require that \( C_i' \) and \( D_i' \) implement the same function. This ensures that the subcircuits \( C_i' \) and \( D_i' \) can be selected independently of the other subcircuits selected.

2) We set an upper bound on the number of inputs to a subcircuit \( C_i' \) that would be considered for replacement. This restricts the complexity of the search for \( C_i' \) and its replacement \( D_i' \).

3) We restrict the structure of the replacement subcircuit \( D_i' \), as explained below. This reduces the complexity of the search for \( D_i' \).

In selecting the structure of the replacement subcircuit \( D_i' \), our main objective is to obtain a structure that is fully testable, has a small number of paths through it, and is area efficient. The structure is introduced in the following section.

III. COMPARISON FUNCTIONS AND COMPARISON UNITS

In this section, we introduce the class of comparison functions. We consider the implementation of a comparison function by a comparison unit that has at most two paths from any one of its inputs to its output. We also consider special cases where the number of paths from certain inputs of a comparison unit to its output is lower than two. We show that comparison units are fully robust-testable for path delay faults. Finally, we consider the identification of comparison functions.

A. Comparison Function Definition and Implementation

Comparison functions are defined as follows.

Definition 1: Let \((x_1, x_2, ..., x_n)\) be a permutation of the input variables \((y_1, y_2, ..., y_n)\) of a function \(f(y_1, y_2, ..., y_n)\). Let the minterms where \(f(x_1, x_2, ..., x_n) = 1\) be \(M = \{m_1, m_2, ..., m_k\}\), where \(m_i\) is given in decimal form. The function \(f(y_1, y_2, ..., y_n)\) is said to be a comparison function if there exists a permutation...
(x₁, x₂, ..., xₙ) and two integers L and U, such that m ∈ M if and only if L ≤ m ≤ U.

The basic building blocks for implementing a comparison function are referred to as comparison blocks. There are two types of comparison blocks, the ≥L block and the ≤U block. A ≥L block produces the output 1 when supplied with an input combination whose decimal value is larger than or equal to L. A ≤U block produces the output 1 when supplied with an input combination whose decimal value is smaller than or equal to U. The implementation of ≥L and ≤U blocks is considered later.

The following example shows how these blocks can be used to implement a comparison function. Note that we use x₄ as the most significant bit and x₁ as the least significant bit in the binary representation of minterms.

Example: Consider the four-input single-output function fₓ(x₁, x₂, x₃, x₄), which is one for minterms {0001, 0101, 0110, 1001, 1010, 1110} or, in decimal form, {1, 5, 6, 9, 10, 14}. Consider the permutation (x₁, x₂, x₃, x₄), where x₁ = y₁, x₂ = y₂, x₃ = y₃, and x₄ = y₄. The function fₓ(x₁, x₂, x₃, x₄) = 1 for the minterms {0101, 0110, 0111, 1000, 1001, 1010}. In decimal form, we have {5, 6, 7, 8, 9, 10}, i.e., all the minterms between five and ten (inclusive). The function can be implemented by the structure shown in Fig. 3. In this implementation, f₂ = 1 if and only if an input combination m is applied, such that 5 ≤ m ≤ 10. In this case, L = 5 = (0101) and U = 10 = (1010).

The structure shown in Fig. 3 is referred to as a comparison unit. An implementation of a ≥L block is shown in Fig. 4(a) for L = (l₁l₂...lₙ). The gate types in Fig. 4(a) are determined as follows.

\[ Gₐ = \begin{cases} \text{AND}, & \text{if } lᵢ = 1 \\ \text{OR}, & \text{if } lᵢ = 0 \end{cases} \]

The gate Gₐ is replaced by a connection to xₙ when lₙ = 1, and replaced by the constant 1 when lₙ = 0. In both cases, Gₐ is omitted. Additional gates may then be omitted. For illustration, the implementation of a ≥L block is shown in Fig. 5(a) for L = 3 = (0011). It can be seen that if x₁ = 1 or x₂ = 1, the input combination represents a number larger than three and the output is one, as required. If x₁ = x₂ = 0, the output is one only if x₃ = x₄ = 1. The implementation of a ≥L block for L = 12 = (1100) is shown in Fig. 5(b). This block demonstrates how the rightmost gates are omitted when the lower bound ends with zeroes. Any input combination where x₁ = x₂ = 1 is larger than or equal to 12 and produces a one output.

An implementation of a ≤U block is shown in Fig. 4(b) for U = (u₁u₂...uₙ). The gate types in Fig. 4(b) are determined as follows.

\[ Gₛ = \begin{cases} \text{AND}, & \text{if } uᵢ = 0 \\ \text{OR}, & \text{if } uᵢ = 1 \end{cases} \]
The gate $G_n$ is replaced by an inverter driven by $x_n$ when $u_n = 0$, and replaced by the constant “1” when $u_n = 1$. Additional gates may then be omitted. For illustration, the implementation of a $\leq U$ block is shown in Fig. 5(c) for $U = 12 = (1100)$. The implementation of a $\leq U$ block for $U = 3 = (0011)$ is shown in Fig. 5(d).

An alternate implementation of a $\leq U$ comparison block can be obtained by observing that the inverted output of a $\geq L$ comparison block implements the function whose minterms are $< L$ (or $\leq L - 1$). Thus, a $\leq U$ comparison block can be implemented by inverting the output of a $\geq (U + 1)$ comparison block. In our procedure, we use the implementation shown in Fig. 4(b).

Next, we consider the number of paths through comparison blocks and comparison units. From Fig. 4, it can be seen that there is at most one path from an input $x_i$ to the output of a comparison block. Comparison units have the structure shown in Fig. 3. Therefore, in a comparison unit, there are at most two paths from any input to the output of the unit. Depending on $L$ and $U$, an input may be omitted altogether from the corresponding comparison block, as shown in Fig. 5(b) and (d). Thus, there may be only one path, or no paths at all, from an input of a comparison unit to its output. Additional special cases exist where the number of paths for some of the inputs is lower than two. These cases are considered in the following subsection.

The longest path through a comparison block has at most $n - 1$ two-input gates, where $n$ is the number of input variables of the comparison function. The longest path through a comparison unit has at most $\eta$ two-input gates. Special cases exist as demonstrated in Fig. 5(b) and (d) and as discussed in Section III-B. In addition, when $k$ consecutive gates are of the same type, they can be combined into a $k$-input gate. During the resynthesis process, the number of logic levels in the circuit can be kept low by considering comparison functions with small numbers of inputs. Experimental results presented in Section V show that the number of logic levels in the resynthesized circuits is approximately the same as in the original circuits, even after technology mapping is applied that avoids gates with large numbers of inputs.

Next, we consider the relationship between comparison functions and threshold functions [15]. Consider a comparison function with input permutation $(x_1, x_2, \ldots, x_n)$ and bounds $L$ and $U$. The $\geq L$ comparison block implements a threshold function with a weight $2^{n-1}$ assigned to $x_i$ and with a threshold value $T = L$. This function would produce an output of one if and only if the corresponding input combination is larger than or equal to $L$ (i.e., $\sum_{i=1}^{n} x_i 2^{n-i} \geq L$), as required. Instead of the $\leq U$ block, it is possible to use a $\geq U + 1$ block that implements a threshold function with the same weights as above and with $T = U + 1$. The complemented output of the $\geq U + 1$ block is then equivalent to the output of a $\leq U$ block. This output and the output of the $\geq L$ block can be ANDed to obtain the required function.

B. Special Cases

In this section, we consider several special cases that result in simplified comparison units.

1) Free Variables: Let $f(x_1, x_2, \ldots, x_n)$ be a comparison function with a lower bound $L = (l_1 l_2 \ldots l_\eta)$ and an upper bound $U = (u_1 u_2 \ldots u_\eta)$. We define the set of free variables as follows.

- **Definition:** $X_F = \{x_1, x_2, \ldots, x_F\}$ is a set of free variables if $l_i = u_i$ for every $1 \leq i \leq F$.

For example, consider a four-input comparison function with $L = 5 = (0101)$ and $U = 7 = (0111)$. In this case, $X_F = \{x_1, x_2\}$. The value of a free variable of a function $f$ is the same for every minterm that sets $f$ to one. As a result, $f$ can be implemented using the structure shown in Fig. 6. The $\geq L_F$ and the $\leq U_F$ blocks have $n - F$ inputs, with $L_F = (l_{F+1} l_{F+2} \ldots l_\eta)$ and $U_F = (u_{F+1} u_{F+2} \ldots u_\eta)$. The free variables drive the output AND gate directly (if their value is one in $L$ and $U$) or through an inverter (if their value in $L$ and $U$ is zero). The number of paths from a free variable to the output of a comparison unit is one.

2) Trivial Lower or Upper Bounds: Let $f(x_1, x_2, \ldots, x_n)$ be a comparison function with a lower bound $L = (l_1 l_2 \ldots l_\eta)$ and an upper bound $U = (u_1 u_2 \ldots u_\eta)$. Let $X_F = \{x_1, x_2, \ldots, x_F\}$ be free variables. Let $m_F$ be the minterm $m$ restricted to the nonfree variables, i.e., $m_F$ is defined over the variables $x_{F+1}, \ldots, x_n$.

Suppose that $L_F = (l_{F+1} l_{F+2} \ldots l_\eta) = (00 \ldots 0)$. Any minterm $m_F$ is larger than or equal to the lower bound $L_F$. In this case, the $\geq L_F$ block reduces to a single line connected to the constant “1,” and the comparison block can be omitted. The number of paths from every input of the comparison unit to its output is at most one in this case.

Suppose that $U_F = (u_{F+1} u_{F+2} \ldots u_\eta) = (11 \ldots 1)$. Any minterm $m_F$ is smaller than or equal to the upper bound $U_F$. In this case, the $\leq U_F$ block reduces to a single line connected to the constant “1,” and the comparison block can be omitted. The number of paths from every input of the comparison unit to its output is at most one.

If $L_F = (00 \ldots 0)$ and $U_F = (11 \ldots 1)$, then the function $f$ can be implemented by a single AND gate driven by the free variables. This case occurs when $f$ has a single prime implicant. For example, let $f(y_1, y_2, y_3) = y_1 y_2$ and let us use the permutation $x_1 = y_1, x_2 = y_3$, and $x_3 = y_2$. Under this permutation, $L = 6$ and $U = 7$. We obtain the set of free variables $X_F = \{x_1, x_2\}$, with $L_F = (0)$ and $U_F = (1)$. Every minterm $m_F$ is between zero and one. Therefore, the function is implemented by a single AND gate driven by $x_1$ and $x_2$ (or $y_1$ and $y_2$).

C. Testability of Comparison Units

In this section, we show that comparison units implemented according to Fig. 6 (i.e., where the free variables drive the output AND gate) are fully robustly testable for path delay faults.
Let $f$ be a comparison function with inputs \{${x_1, x_2, \ldots, x_n}$\}. $L = (l_1 l_2 \cdots l_n)$ and $U = (u_1 u_2 \cdots u_n)$, and let the set of free variables be $X_F = \{x_1, x_2, \ldots, x_F\}$.

To describe two-pattern tests for path delay faults, we use the values 000 and 111 to denote stable zero and one values, respectively; and we use 0x1 and 1x0 to denote rising transitions and falling transitions, respectively. We demonstrate the construction of a complete test set by the following example. The proof that every comparison unit is fully robustly testable for path delay faults is given below.

**Example:** Consider the comparison unit shown in Fig. 7. In this case, $L = 11$, $U = 12$, $X_F = \{x_1\}$, $L_F = 3$, and $U_F = 4$.

To test the path delay faults starting from $x_1$, we set $x_1 = 1x0$ or $x_1 = 0x1$ (depending on the fault). To set the other two inputs of the output AND gate to 111, we apply to $(x_2, x_3, x_4)$ a stable input combination between three and four. In this example, we apply three, or (000, 111, 111). To test the path delay faults starting from $x_2$ and going through the $\geq L_F$ block, we set $x_2 = 0x1$ or $x_2 = 1x0$. To propagate the fault, we set $x_3 = 0x4 = 0x0$. This corresponds to the smallest possible decimal value that propagates the transition on $x_3$ to the output. It results in the output of the $\geq U_F$ block being 111 (larger decimal values may not yield 111 on the output of the $\geq U_F$ block). In addition, we set $x_1 = 111$. To test the path delay faults starting from $x_3$ and going through the $\geq L_F$ block, we set $x_3 = 0x0$ or $x_3 = 1x0$. To propagate the fault, we set $x_2 = 000$ and $x_4 = 111$. This results in the output of the $\leq U_F$ unit being 111. In addition, we set $x_1 = 111$. The remaining tests are derived in a similar way.

The complete test set is shown in Table I.

Next, we consider path delay faults through the $\geq L_F$ block. The path delay faults through the $\leq U_F$ block can be treated in a similar way.

We distinguish between two types of path delay faults through a $\geq L_F$ block: those that start from inputs driving an AND gate and those that start from inputs driving an OR gate. In every case, we show that the two-pattern input combination $(T_1, T_2)$ that propagates the fault through the $\geq L_F$ block satisfies $T_1 \leq U_F$ and $T_2 \leq U_F$. Therefore, the output of the $\leq U_F$ block is one for both $T_1$ and $T_2$. In addition, we show that the output of the $\leq U_F$ does not change between $T_1$ and $T_2$. Consequently, its value is 111, allowing the fault to propagate to the output of the comparison unit.

Consider an input $x_i$ of the $\geq L_F$ block, driving an AND gate $G$. Consider the test where input $x_j$ is assigned the value $l_j j_j$ for every $j > F$, $j \neq i$. In the same test, input $x_i$ is assigned a 1x0 or 0x1 transition, depending on the fault type. By assigning $x_j = l_j j_j$ to every $j < i$, the fault effect is propagated through the gates to the left of gate $G$. By assigning $x_j = l_j j_j$ for every $i < j \leq n$, the other input of the AND gate driven by $x_i$ is set to 111, allowing the fault to be activated. Thus, we obtain $T_1 = (l_{F+1} \cdots l_{i-1} 1 l_{i+1} \cdots l_n)$ and $T_2 = (l_{F+1} \cdots l_{i-1} 0 l_{i+1} \cdots l_n)$ (or vice versa). Since we assume that $x_2$ drives an AND gate, $l_2 = 1$. Therefore, both $T_1$ and $T_2$ do not exceed $L_F$, and thus, they do not exceed $U_F$. To show that the output of the $\leq U_F$ block does not change between $T_1$ and $T_2$, observe that for the first nonfree variable $x_{F+1}$, we must have $l_{F+1} = 0$ and $u_{F+1} = 1$ ($l_{F+1} = u_{F+1}$ would imply that $x_{F+1}$ is also a free variable; and the combination $l_{F+1} = 1$ and $u_{F+1} = 0$ is impossible since $L_F \leq U_F$). Thus, the leftmost gate in the $\geq L_F$ block and in the $\leq U_F$ block is an OR gate. Therefore, $x_{F+1} = 0$ in both $T_1$ and $T_2$, and due to the inverter on $x_{F+1}$ in the $\leq U_F$ block, we obtain a stable value 111 at the output of the $\leq U_F$ block.

Next, consider an input $x_i$ of the $\geq L_F$ block, driving an OR gate. Consider the test where input $x_i$ is assigned a 1x0 or 0x1 transition, depending on the fault type, and input $x_j$ is assigned the value $l_j j_j$ for every $j < i$. This allows propagation of the fault effect from input $x_i$ to the output of the $\geq L_F$ block. In addition, we need to set the other input of the OR gate driven by input $x_i$ to 000. To achieve this objective, we set $x_j = 000$ for every $i < j \leq n$. Thus, we obtain $T_1 = (l_{F+1} \cdots l_{i-1} 10 \cdots 0)$ and $T_2 = (l_{F+1} \cdots l_{i-1} 00 \cdots 0)$ (or vice versa). If $i \neq F+1$, then we showed above that $l_{F+1} = 0$ results in the value 111 at the output of the $\leq U_F$ block. If $i = F+1$, then $l_{F+2} = \cdots = l_n = 0$ results in the value 111 on the other input of the leftmost OR gate in the $\leq U_F$ block (due to the inverter and the fact that the rightmost gate is driven by two inverted inputs). Thus, again, we have the value 111 at the output of the $\leq U_F$ block. This value allows the fault to propagate to the output of the comparison unit.

### TABLE I

<table>
<thead>
<tr>
<th>fault</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>111</td>
<td>01x0</td>
<td>000</td>
<td>111</td>
</tr>
<tr>
<td>$x_2$</td>
<td>111</td>
<td>000</td>
<td>01x0</td>
<td>111</td>
</tr>
<tr>
<td>$x_3$</td>
<td>111</td>
<td>111</td>
<td>01x0</td>
<td>111</td>
</tr>
<tr>
<td>$x_4$</td>
<td>111</td>
<td>111</td>
<td>000</td>
<td>1x0</td>
</tr>
</tbody>
</table>

D. Identifying Comparison Functions

We use an exhaustive procedure that enumerates the permutations of the variables of a function $f(y_1, y_2, \ldots, y_n)$ in order to determine whether $f$ is a comparison function. For every permutation $(x_1, x_2, \ldots, x_n)$ of the variables of $f$, we obtain the minterms where $f(x_1, x_2, \ldots, x_n) = 1$. If all these minterms
TABLE II
EXHAUSTIVE SEARCH FOR COMPARISON FUNCTIONS

<table>
<thead>
<tr>
<th>f3</th>
<th>y1</th>
<th>y2</th>
<th>y3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>0 0 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>0 1 0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>0 1 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>1 0 0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>1 0 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>1 1 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>1 1 1 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For example, we show in Table II a function \( f_3(y_1, y_2, y_3) \) and the same function under all the permutations of its inputs. By considering \( f_3(y_1, y_2, y_2) \) or \( f_3(y_3, y_2, y_2) \), it can be seen that the function is a comparison function.

The complexity of this procedure is \( O(n! \cdot 2^n) \), where \( n \) is the number of variables of \( f \). The term \( n! \) results from the worst case where all \( n! \) permutations of the variables of \( f \) have to be considered. The term \( 2^n \) results from considering the minterms where \( f = 1 \).

To consider functions with larger numbers of variables, it is necessary to reduce the complexity of identifying comparison functions. To remove the \( n! \) complexity term, the method developed in [20] can be used. It is also possible to consider incompletely specified cubes over the values \( \{0, 1, x\} \), instead of considering fully specified minterms. This can help reduce the \( 2^n \) complexity term. We do not pursue these options here, since the results of this paper and the results presented in [20] indicate that it is sufficient to consider comparison functions with small numbers of inputs for resynthesis to yield significant reductions in the numbers of paths.

IV. RESYNTHESIS TO REDUCE THE NUMBER OF PATHS USING COMPARISON FUNCTIONS

In this section, we consider the problem of reducing the number of paths in a given gate-level circuit by using comparison functions. We propose two procedures. In the first procedure, our goal is to reduce the number of paths without increasing the number of gates. In the second procedure, our goal is to reduce the number of paths even if it increases the number of gates. The second procedure yields circuits with smaller numbers of paths, at the cost of a larger number of gates.

A. Reducing the Number of Paths Without Increasing the Number of Gates

The general form of the resynthesis procedure that reduces the number of paths without increasing the number of gates is as follows. The procedure traces the circuit from the primary outputs toward the primary inputs. Every gate output \( g \) in the circuit is considered, except for gate outputs that become internal to comparison units. For every gate output \( g \) considered, we derive several subcircuits with output \( g \) as explained later (following Procedure 1). For each subcircuit \( C' \) with set of inputs \( I' \), we check whether the function \( f'(I') \) that \( C' \) implements at \( g \) is a comparison function. If \( f' \) is not a comparison function, \( C' \) is discarded. Otherwise, we compute the number of gates \( N' \) in a comparison unit implementing \( f' \). We compare this number to the number of gates currently implementing \( g \), denoted by \( N \). In computing \( N \), we take into account the fact that some of the gates may fan out, and thus may be common to \( f' \) and to other subfunctions. Such common gates are not included in the count \( N \), since they cannot be removed even if \( C' \) is replaced by a comparison unit. We then select the comparison unit that results in the largest reduction in the number of gates and replace the corresponding subcircuit \( C' \) by it. If a choice exists, we select the comparison unit that results in the smallest number of paths on \( g \).

Procedure 1: Reducing the number of paths without increasing the number of gates.

1) Mark all the primary outputs and unmark all other lines.

Set \( g = N_L \), where \( N_L \) is the number of circuit lines.
2) If line $g$ is a marked gate output:
   a) find all the candidate subcircuits with output $g$. For every candidate subcircuit $C'$:
      i) if $C'$ does not implement a comparison function, eliminate $C'$;
      ii) otherwise, compute the reduction $N - N'$ obtained in the number of gates if $C'$ is replaced by a comparison unit.
   b) for every subcircuit $C'$ that results in the maximum reduction in the number of gates, compute the number of paths from the primary inputs to $g$ if $C'$ is replaced by a comparison unit (the computation method follows the one described in Section II);
   c) select the subcircuit $C'$ that results in the maximum reduction in the number of gates and the minimum number of paths from the primary inputs to $g$;
   d) mark that $C'$ is selected and mark all the inputs of $C'$ which are not primary inputs [such lines will be considered in Step 2).

3) Set $g = g + 1$. If $g > 0$, go to Step 2).

Next, we describe the computation of all the candidate subcircuits with a given output $g$. The computation starts with the subcircuit $C_0$ containing the gate with line $g$ as its output. Let the $i$th subcircuit obtained be $C_i$. Let $I_i$ be the set of inputs of $C_i$. For every line $h \in I_i$ such that $h$ is the output of gate $H$, we define a subcircuit $C_k = C_i \cup \{H\}$. If the number of inputs of $C_k$ does not exceed a predetermined limit $K$, $C_k$ is also used for generating new subcircuits. The process ends when no new subcircuits can be generated. For example, in Fig. 8, we have $C_0 = \{G_1\}$, $C_1 = \{G_1, G_2\}$, $C_2 = \{G_1, G_3\}$, $C_3 = \{G_1, G_2, G_3\}$, and so on. Values of $K = 3, 6$ were found to be useful in our experiments.

After Procedure 1 is applied, a new gate-level circuit is generated by replacing every subcircuit $C'$ selected in Step 2c) of Procedure 1 with its comparison unit implementation. Procedure 1 is then applied to the new circuit. This is repeated until no further reduction in the number of gates is obtained. Repeating Procedure 1 is also motivated by the fact that the number of paths in the original circuit (the same labels are used in Step 2b) of Procedure 2) is computed based on the labels assigned by the path counting procedure of [9] to the original circuit, as explained in the following subsection.

The worst case complexity of Procedure 1 can be determined as follows. In the worst case, Procedure 1 considers every gate output. Denoting the number of gates by $N_G$, Procedure 1 goes through $N_G$ iterations. The complexity of each iteration is determined by the need to find candidate subcircuits and evaluate whether they implement comparison functions. We denote the number of subcircuits for a gate output $g$ by $N_{subc}$. The value of $N_{subc}$ may be exponential in the number of circuit gates $N_G$. However, in practice, we limit the number of subcircuits by imposing a limit on the number of inputs to each subcircuit. For every subcircuit, we first need to obtain the function implemented by the subcircuit. Denoting the number of subcircuit inputs by $K$ and the number of gates in a subcircuit by $N_{SG}$, the complexity of this step is $O(N_{SG}2^K)$. In addition, we need to determine whether the subcircuit implements a comparison function. This step has complexity $O(K2^K)$. Since we consider small values of $K$, we can consider $2^K$ and $K2^K$ as constants. We obtain $O(N_GN_{SG}N_{subc})$ for the overall complexity of Procedure 1.

B. Reducing the Number of Paths

The general form of the resynthesis procedure that reduces the number of paths while allowing the number of gates to be increased is similar to Procedure 1 and is given next.

Procedure 2: Reducing the number of paths.

1) Label the circuit using the procedure from [9]. Mark all the primary outputs and unmark all other lines. Set $g = N_L$.

2) If line $g$ is a marked gate output:
   a) Find all the candidate subcircuits with output $g$. For every candidate subcircuit $C'$:
      i) if $C'$ does not implement a comparison function, eliminate $C'$;
      ii) otherwise, compute the number of paths from the primary inputs to $g$ if $C'$ is replaced by a comparison unit.
   b) Select the subcircuit $C'$ that results in the minimum number of paths from the primary inputs to $g$. Mark that $C'$ is selected and mark all the inputs of $C'$ that are not primary inputs.

3) Set $g = g + 1$. If $g > 0$, go to Step 2).

We do not use the number of gates as a secondary objective in Procedure 2, since our experimental results indicate that it does not improve the results obtained.

We point out that a subcircuit $C'$ is selected for a line $g$ in Step 2b) of Procedure 2 based on the labels on the inputs of $C'$ in the original circuit (the same labels are used in Step 2b) of Procedure 1). If later the labels on the inputs of $C'$ change due to changes in the subcircuits that drive $C'$, we do not consider $C'$ again. To compensate for this effect, we apply Procedure 2 repeatedly, producing new circuits to which Procedure 2 is applied again, until no improvement in the number of paths is obtained. Several iterations of Procedure 2 were required before no improvement was possible for the benchmark circuits we considered.

The worst case complexity of Procedure 2 is the same as that of Procedure 1.

V. EXPERIMENTAL RESULTS

In this section, we describe the results of resynthesis by Procedures 1 and 2 and then demonstrate the effects of resynthesis on testability.

A. Resynthesis

We applied Procedures 1 and 2 to irredundant, fully scanned ISCAS-89 benchmark circuits that have more than 10,000 paths. Irredundant circuits were obtained using the procedure of [16]. Comparison functions were identified by trying up to 200 permutations of the inputs and checking whether either the minterms where the function is one or the minterms where
the function is zero are consecutive. In the latter case, \( f \) was implemented as a comparison function and \( f \) was obtained by complementing the comparison unit output.

To measure the change in the number of gates due to Procedures 1 and 2, we count the number of equivalent two-input gates. A \( k \)-input gate can be implemented as an interconnection of \( k-1 \) two-input gates, and therefore adds \( k-1 \) to the gate count. We used equivalent two-input gates to ensure that the way in which gates with large numbers of inputs are implemented does not affect the gate count (Procedure 1 also uses the number of equivalent two-input gates as a selection criterion). For all the circuits, we considered subcircuits with up to \( K = 5 \) and up to \( K = 6 \) inputs (in two separate experiments) to identify comparison functions. We also considered some of the smaller circuits with \( K = 7 \); but in the majority of cases, the results obtained were inferior to the results obtained with \( K = 5 \) and \( K = 6 \). Low values of \( K \) have the advantage that they result in comparison units having small numbers of logic levels. Consequently, the total circuit delay is likely to remain the same.

Several circuits turned out to contain redundant stuck-at faults after applying Procedure 1. This is in spite of the fact that the original circuits are irredundant and that comparison units are fully testable for stuck-at faults when their inputs are independently controlled. The reason for the existence of redundant faults is as follows. Consider a subcircuit \( C \) with a set of inputs \( I \), replaced by a comparison unit \( D \) with the same set of inputs \( I \). Suppose that a stuck-at fault \( f \) in \( D \) requires a combination \( \mathbf{\alpha} \) on \( I \). Suppose in addition that \( \mathbf{\alpha} \) is not required on \( I \) for any fault in \( C \). If \( \mathbf{\alpha} \) cannot be obtained on \( I \) due to the logic driving it, then \( f \) is redundant in the modified circuit even if the original circuit was irredundant. The redundancy removal procedure from [16] was applied after Procedure 1 whenever redundant faults were found.

The results of Procedure 1 followed by redundancy removal are reported in Table III in the following format. After the circuit name, we give the value of \( K \) for which the modified circuit with the lowest number of paths was obtained. We then give the number of primary inputs and the number of primary outputs. The numbers of two-input gates in the original irredundant circuit, in the modified circuit, and in the modified circuit after redundancy removal are given next. In the last column, we give the number of paths in the original irredundant circuit, in the circuit after modification, and in the modified circuit after redundancy removal. The results after redundancy removal are omitted if no redundant stuck-at faults were found. It can be seen that the proposed procedure consistently reduces the number of paths without increasing the number of gates. The reduction in the number of paths is often very large. Redundancy removal has a minor effect on the size of the circuit; however, it is important to ensure complete stuck-at fault testability.

We also applied the technology mapping procedure included in SIS to the circuits before and after resynthesis based on Procedure 1. We used the rugged script of SIS. The library for technology mapping includes inverters, two to four input NAND gates, two to four input NOR gates, and the constants zero and one. Before applying technology mapping, we made the following changes to arrive at a circuit representation that is as uniform as possible. We removed buffers, single-input AND gates, and single-input OR gates; we removed chains of inverters containing even numbers of inverters; if multiple fanout branches \( g_1, g_2, \ldots, g_k \) of the same stem have inverters on them, we defined a single branch \( g \) with an inverter on it, removed the inverters from \( g_1, g_2, \ldots, g_k \) and connected them to \( g \); and we removed primary inputs that are also primary outputs. Circuit parameters are shown in Table IV. For each set of circuits, we show the number of literals and the number of gates on the longest path (indicating the circuit delay). It can be seen that reductions in circuit size are consistent with those of Table III, although the contents of the technology library was not directly considered by the proposed resynthesis procedure. In addition, the length of the longest path through the circuit does not increase with the application of the procedure proposed here.

The results of applying Procedure 2, which targets the reduction of the path count more aggressively than Procedure 1, are shown in Table V in the following format. After the circuit name and the value of \( K \) for which the results were obtained, we give the number of primary inputs and the number of primary outputs. The number of two-input gates in the original and in the modified circuits are given next. We then give the number of paths in the original circuit and in the modified circuit. For all the circuits, we considered subcircuits with up to \( K = 5 \) and up to \( K = 6 \) inputs (in two separate experiments). The best result obtained for each circuit is reported. It can be seen that the number of paths is reduced more than in Table III. However, the path reduction is sometimes achieved at the cost of increasing the number of gates.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>( K )</th>
<th>inp</th>
<th>out</th>
<th>2-input gates</th>
<th>orig</th>
<th>modif</th>
<th>red.rem</th>
<th>orig</th>
<th>modif</th>
<th>red.rem</th>
</tr>
</thead>
<tbody>
<tr>
<td>irs1423</td>
<td>6</td>
<td>91</td>
<td>79</td>
<td>491 490</td>
<td>488</td>
<td>42,089</td>
<td>37,293</td>
<td>37,278</td>
<td></td>
<td></td>
</tr>
<tr>
<td>irs5378</td>
<td>6</td>
<td>214</td>
<td>224</td>
<td>1394 1358</td>
<td>10,976</td>
<td>10,581</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irs9234</td>
<td>6</td>
<td>247</td>
<td>248</td>
<td>1929 1784</td>
<td>1783</td>
<td>109,283</td>
<td>20,333</td>
<td>20,330</td>
<td></td>
<td></td>
</tr>
<tr>
<td>irs13207</td>
<td>6</td>
<td>699</td>
<td>788</td>
<td>2737 2537</td>
<td>261,312</td>
<td>85,174</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irs15850</td>
<td>6</td>
<td>611</td>
<td>680</td>
<td>3361 3115</td>
<td>3107</td>
<td>25,003,369</td>
<td>3,655,332</td>
<td>3,584,511</td>
<td></td>
<td></td>
</tr>
<tr>
<td>irs35932</td>
<td>5</td>
<td>1763</td>
<td>2048</td>
<td>9900 8497</td>
<td>58,645</td>
<td>20,898</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>irs38417</td>
<td>5</td>
<td>1664</td>
<td>1742</td>
<td>9698 9344</td>
<td>9316</td>
<td>1,192,971</td>
<td>674,081</td>
<td>672,121</td>
<td></td>
<td></td>
</tr>
<tr>
<td>irs38584</td>
<td>5</td>
<td>1455</td>
<td>1700</td>
<td>12037 11773</td>
<td>565,433</td>
<td>157,979</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Circuit</th>
<th>original</th>
<th>Proc.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>cir1423</td>
<td>1,035 72</td>
<td>1031 70</td>
</tr>
<tr>
<td>irs5378</td>
<td>2607 17</td>
<td>2610 16</td>
</tr>
<tr>
<td>irs9234</td>
<td>3817 30</td>
<td>3577 30</td>
</tr>
<tr>
<td>irs13207</td>
<td>5443 31</td>
<td>5004 31</td>
</tr>
</tbody>
</table>
TABLE V
RESULTS OF PROCEDURE 2

<table>
<thead>
<tr>
<th>circuit</th>
<th>K</th>
<th>inp</th>
<th>out</th>
<th>2-inp gates</th>
<th>paths</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>orig</td>
<td>modif</td>
<td>orig</td>
<td>modif</td>
</tr>
<tr>
<td>irs1423</td>
<td>6</td>
<td>91</td>
<td>79</td>
<td>491</td>
<td>40,893</td>
</tr>
<tr>
<td>irs5378</td>
<td>6</td>
<td>214</td>
<td>224</td>
<td>1394</td>
<td>10,976</td>
</tr>
<tr>
<td>irs9234</td>
<td>6</td>
<td>247</td>
<td>248</td>
<td>1929</td>
<td>109,283</td>
</tr>
<tr>
<td>irs13207</td>
<td>6</td>
<td>699</td>
<td>788</td>
<td>2737</td>
<td>261,312</td>
</tr>
<tr>
<td>irs15580</td>
<td>6</td>
<td>611</td>
<td>680</td>
<td>3361</td>
<td>23,003,369</td>
</tr>
<tr>
<td>irs35932</td>
<td>5</td>
<td>1763</td>
<td>2048</td>
<td>9900</td>
<td>58,645</td>
</tr>
<tr>
<td>irs38417</td>
<td>6</td>
<td>1664</td>
<td>1742</td>
<td>9698</td>
<td>1,192,971</td>
</tr>
<tr>
<td>irs38584</td>
<td>6</td>
<td>1455</td>
<td>1700</td>
<td>12139</td>
<td>565,433</td>
</tr>
</tbody>
</table>

**B. Testability**

In this section, we report the effect of the proposed resynthesis procedure on the testability of the resulting circuits. We consider their random pattern testability for stuck-at faults and for path delay faults. We also consider the path delay fault coverage obtained by a deterministic procedure.

1) Stuck-At Faults: To determine the stuck-at fault testability, we applied up to 30,000,000 random patterns to the circuits obtained by Procedure 1. We used the fault simulator FSIM [18] to obtain the fault coverage. The results are reported in Table VI as follows. After the circuit name, we give the number of stuck-at faults, the number of faults that remained undetected after applying 30,000,000 random patterns, and the last pattern that was effective in detecting any fault in the original circuit. Next, we give the same information for the modified circuit after redundancy removal. It can be seen that the random pattern testability for stuck-at faults remained unchanged after the modifications. This is important since some resynthesis procedures may cause the random pattern testability to deteriorate [17].

2) Path Delay Faults: In this section, we consider the robust path delay fault testability of the circuits.

We applied 100,000 random two-pattern tests to each circuit and recorded the number of path delay faults detected robustly. The number of faults detected was plotted as a function of the number of random patterns applied. The results for *irs 13 207* are shown in Fig. 9. The trend shown by the curves of Fig. 9 continued until the number of detected faults saturated. Similar graphs were obtained for other circuits as well. It can be seen that the number of path delay faults detected in the modified circuit is larger than in the original circuit, and the difference in the number of detected faults between the modified and the original circuits increases as the number of random patterns increases. This is in addition to the reduction in the number of paths from 261,312 to 85,174, or the reduction from 522,624 to 170,348 in the number of path delay faults for *irs 13 207* (the number of path delay faults is twice the number of paths).

The total number of faults are given in this order under column det./faults. All the circuits show an increase in the number of detected path delay faults and a reduction in the total number of path delay faults, resulting in an improvement of the path delay fault coverage. For example, for *irs 38 584*, resynthesis increases the number of detected faults from 46,189 to 48,037 while reducing the total number of faults from 1,130,866 to 315,958.

We also computed the path delay fault coverage achieved by the deterministic test generation procedure of [19]. Three sets of circuits were considered: the original circuits, the circuits obtained by Procedure 1 (see Table III), and the circuits obtained by Procedure 2 (see Table V). The procedure of [19] finds faults that do not need to be tested (called RD-faults) according to the definition of [12]. It then finds robust tests for the remaining (non-RD) faults. If all non-RD faults are robustly tested, then the correct temporal behavior of the circuit is guaranteed and the remaining faults do not need to be tested [12].

The robust fault coverage reported by the procedure of [19] is computed as the number of robustly tested faults, divided by the total number of non-RD faults. The results for the three sets of circuits are shown in Table VIII. After the circuit name, we show in Table VIII the robust fault coverage for the original circuit, for the circuit obtained by using Procedure 1, and for the circuit obtained by using Procedure 2. It can be seen that Procedures 1 and 2 are effective in increasing the testability of the circuit, especially for the three largest circuits. Procedure 2 gives some improvement in fault coverage compared to Procedure 1, and
VI. Generalized Comparison Functions

We defined a comparison function as a function for which there exist a permutation \( X = (x_1, x_2, \ldots, x_n) \) of the input variables and constants \( L \) and \( U \) such that every minterm \( m \) under the permutation \( X \) satisfies \( L \leq m \leq U \). Given an arbitrary function \( f \), it is possible to partition the set of minterms of \( f \) into subsets \( M_1, M_2, \ldots, M_k \) such that the function \( f_i = \sum \{m; m \in M_i\} \) is a comparison function. The function \( f \) can then be implemented as \( f = f_1 + f_2 + \cdots + f_k \), where \( f_i \) is a comparison function for every \( 1 \leq i \leq k \).

In this case, \( f \) can be implemented as \( p_c g_c + p_1 g_1 + p_2 g_2 \) where \( p_1 \) and \( p_2 \) are products of literals determined by \( X_F \), \( p_c = p_1 + p_2 \), and the functions \( g_c, g_1, \) and \( g_2 \) are \( g_c = \sum \{m; L_F \leq m \leq U_F \}, g_1 = \sum \{m; L_F \leq m \leq L_{F_2} \}, \) and \( g_2 = \sum \{m; U_F \leq m \leq L_{F_2} \} \).

We counted the number of subcircuits considered by Procedure 1 that implement comparison functions and that implement double comparison functions with overlap. We checked whether a subcircuit implements a double comparison function with overlap only if it turned out that it does not implement a comparison function. We found that large numbers of subcircuits exist that do not implement comparison functions but implement double comparison functions with overlap. For example, for \( s9234 \) with \( K = 5 \), we found 6882 single comparison functions and 1586 double comparison functions. Using \( K = 6 \), we found 9173 single comparison functions and 3334 double comparison functions.

VII. Concluding Remarks

We defined a class of functions called comparison functions and showed how they can be implemented using comparison units. Comparison units are fully testable for stuck-at faults and path delay faults. In addition, comparison units have a small number of paths through them. We proposed a method of modifying a given circuit to enhance its path delay fault testability by replacing subcircuits of the given circuit with comparison units. This could be achieved without increasing the number of gates in the circuit or its longest path. Experimental results showed orders of magnitude reductions in the number of paths in the circuits considered. It was shown that most of the path delay faults again shows a significant improvement in testability compared to the original circuit.
removed were untestable by random patterns, improving the random path delay fault testability of the circuit significantly. The random pattern testability of the circuits to stuck-at faults remained unchanged.

Several extensions of the proposed resynthesis procedure are possible. 1) Combinations of values that cannot be obtained due to logic dependencies in the circuit can be used during the selection of comparison units to further reduce the number of paths and to ensure that all the faults in each comparison unit can be tested. 2) Resynthesis using generalized comparison units can help reduce the number of paths even further.

REFERENCES


