Test Scheduling and Control for VLSI Built-In Self-Test

GARY L. CRAIG, MEMBER, IEEE, CHARLES R. KIME, SENIOR MEMBER, IEEE, AND KEWAL K. SALUJA, MEMBER, IEEE

Abstract—**The problem of exploiting parallelism in the testing of VLSI circuits with built-in self-test (BIST) was first introduced in [1]. In this paper, this problem is examined in detail using a broader modeling foundation and new algorithms. A hierarchical model for VLSI circuit testing is introduced. The test resource sharing model from [1] is employed to exploit the potential parallelism. Based on this model, very efficient suboptimum algorithms are proposed for defining test schedules for both the equal length test and unequal length test cases. For the unequal length test case, three different scheduling disciplines are defined and scheduling algorithms are given for two of the three cases. Data on algorithm performance are presented. The issue of the control of the test schedule is also addressed, and a number of structures are proposed for implementation of control.**

Index Terms—**Built-in self-test, cliques, design for testability, graph coloring, scheduling, test control, VLSI circuit testing.**

I. Introduction

IN VERY large scale integrated (VLSI) circuits, there exists a large device count and a relatively few input/output pins. This can produce complex structures for which test generation is difficult and results in long tests with high input/output traffic during testing. One approach to dealing with this difficult testing problem is to employ built-in self-test (BIST) [2]. In the VLSI environment, desirable goals for BIST are to

1) eliminate as much test generation as possible,
2) permit a fairly general class of failure modes,
3) permit easy circuit initialization and observation,
4) reduce input/output pin signal traffic, and
5) reduce test length.

Although BIST techniques clearly realize a number of the goals listed, for very large circuits with extensive BIST resources, the testing time can still be quite long if the tests for the various parts of the circuits are executed one after the other. In such cases, in order to reduce testing time and fully exploit the power of the BIST resources, it is essential to control the testing process so that full use is made of the potential parallelism available.

In order to develop a perspective for parallelism in BIST, consider the testing of a block of logic within a VLSI chip. The inputs to the block under test (BUT) must be stimulated with an appropriate input sequence including initialization steps. The outputs of the BUT must be observed and the response analyzed to determine if the block is faulty or not. The observation of the response must typically be coordinated with the application of the input sequence.

In the typical BIST implementation for testing a block of logic, the original source of the stimuli is a set of one or more test pattern generators (TPG) and the final destination of the responses is a set of one or more compressors and/or response analyzers [3]. It is possible that the test generators and response compressors and/or analyzers are directly attached to the block under test. Often, however, there is additional logic lying between the test generators and the BUT and between the BUT and the response compressors/analyzers. Thus, test control logic must exist which controls not only the test pattern generators and response compressors/analyzers but also this intervening logic. Typically, paths must be established from the test pattern generators to the inputs of the BUT and from the BUT to the response compressors/analyzers. In addition, the test control logic must interact with a higher level of control either on or off the chip. The blocks which are required to perform a test (test control logic, TPG's, compressors/analyzers, BUT, and any intervening logic) are known as *test resources*. Test resources may be shared among BUT's. For example, testing schemes exist in which the response compressor for one BUT can be used as an input stimulus, i.e., as a TPG, for another BUT [3]. Also, for those blocks which lie on the periphery of the chip, a portion of the test resources may lie off-chip. For a block entered by primary inputs, all or part of a TPG may lie off-chip. For a block feeding primary outputs, all or part of a response compressor/analyzer may likewise lie off-chip.

In this paper, the potential for parallel execution of tests will be exploited. Models, including one based on test resources, will be developed, algorithms for generating test schedules will be given, and the control of resources in executing the test schedule will be explored.

II. A Hierarchical Model for Test Parallelism

The approach to testing discussed in Section I and the notion of a test as an element of a hierarchy can be used to develop a model for parallelism in the self-test of an integrated circuit. The resulting model is hierarchical in nature due to the underlying design of the circuit and the relationship between the tests and the test resources. It should be noted that two major types of test parallelism problems have been identified in the literature thus far [1], [4]. One of the forms as discussed in [1] deals with tests for blocks of logic; these tests which we refer to as *block tests* potentially consist of many test vectors

Manuscript received July 15, 1986; revised April 17, 1987. This work was supported in part by the National Science Foundation under Grants DCR-8206564 and DCR-8509194 and by the Australian Department of Science and Technology.

G. L. Craig is with the Department of Electrical and Computer Engineering, Syracuse University, Syracuse, NY 13215.

C. R. Kime and K. K. Saluja are with the Department of Electrical and Computer Engineering, University of Wisconsin, Madison, WI 53706.

IEEE Log Number 8718428.

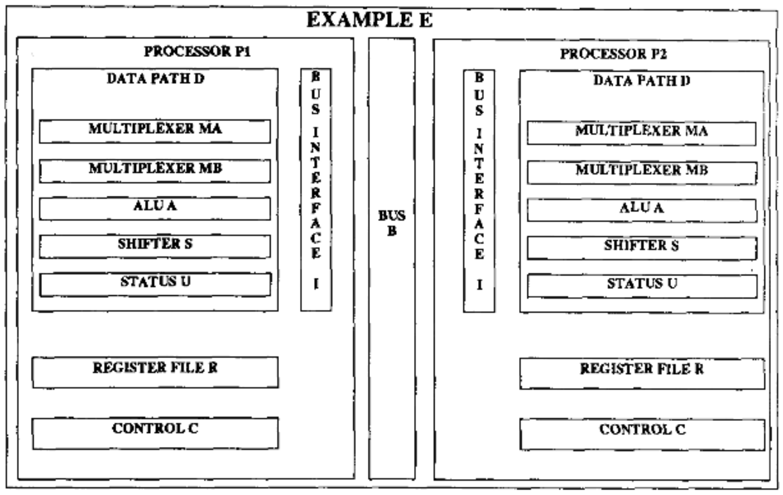

Fig. 1. Block diagram of example illustrating testing hierarchy.

and are regarded as indivisible entities for scheduling. Furthermore, there is no temporal relationship between the test vectors in different block tests other than that defined by conflicting use of resources. The other form, as discussed in [4], deals with *test steps* which may need to appear in and utilize resources in a specific temporal order. Although certain aspects of the model in this section may apply to both forms, the remaining sections of the paper will deal exclusively with scheduling of block tests.

To provide a general perspective for overall parallelism, an example and the corresponding hierarchical model for test scheduling is presented. The example circuit to be considered is shown in block diagram form in Fig. 1. The circuit contains a bus B and two 8-bit processors, $P1$ and $P2$, which are not necessarily identical. Each processor contains, as detailed in Fig. 1, a data path D, a register file R, a control C, and a bus interface I. Each data path, in turn, contains two multiplexers MA and MB, an ALU A, a shifter S, and a status unit U.

Suppose that the testing of this example circuit is considered from a hierarchical view with parallelism of testing in mind. We begin with the testing of the components of the data path and have tests t_{MA}, t_{MB}, t_A, t_S, and t_U for the multiplexers MA and MB, ALU A, shifter S, and status U, respectively. In the design of the BIST hardware for the data path, the designer realizes that it is possible to perform test t_{MA} and t_{MB} simultaneously but that due to sharing of hardware the remaining tests must be executed serially. Thus, t_{MA} and t_{MB} are scheduled simultaneously followed by t_A, t_S, and t_U. This scheduled set of tests is referred to by the designer as t_D, the data path test. Similarly, tests can be obtained for the register file, the control, and the bus interface. These tests are t_R, t_C, and t_I, respectively. This can be done for the two processors and the bus as well. Thus, at the second level of the hierarchy, there are eight tests t_{D1}, t_{R1}, t_{C1}, t_{I1}, t_{D2}, t_{R2}, t_{C2}, and t_{I2}. In terms of the design partition of the hardware, a third level of tests can be established, t_{P1}, t_{P2}, and t_B. We can choose to schedule tests at each level of the hierarchy or can combine levels in the model to be proposed. For the example, we will combine in a natural way levels two and three. In general, such a combination of levels prior to scheduling can lead to more efficient schedules. A formal model for hierarchical test schedules follows and is illustrated via this example.

The overall test schedule for the circuit can be viewed as a partial order in which $t_i \leq t_j$ if either $t_i = t_j$ or t_i is scheduled before t_j. Furthermore, universal bounds can be defined for this partial order as tests B (begin) and E (end) which may correspond to test control functions or may simply be null tests. The test schedule can then be represented by a Hasse diagram [5] which is a single entry point–single exit point acyclic directed graph. However, it is also possible to represent any test which has internal scheduling in the hierarchy with the same structure. When detail is desired, the graph for a test can simply replace its single node because of the single entry point–single exit point structure. To avoid ambiguity, if there is more than one appearance of B and E in a graph that represents multiple levels in the hierarchy, the appearances of B and E will be subscripted. Note that the graph representing a test that has not been internally scheduled consists simply of disconnected nodes representing the subtests.

The test structure developed thus far for the example is partially represented in Fig. 2 using the partial order and hierarchy. The schedule for testing of the data path $D1$ is shown with MA and MB being tested in parallel. At the next level, the tests are at present unscheduled, so there are no arcs in the graph. At the top of the model, there is a single node representing the testing of the circuit. Later, after developing the theory and methods for test scheduling, a schedule for the combined second and third levels in the example model will be found, thus completing the schedule for the circuit test.

Thus far, a hierarchical model for test schedules has been given. In the next section, a model which will serve as a foundation for obtaining a schedule for an unscheduled portion of the hierarchy will be developed.

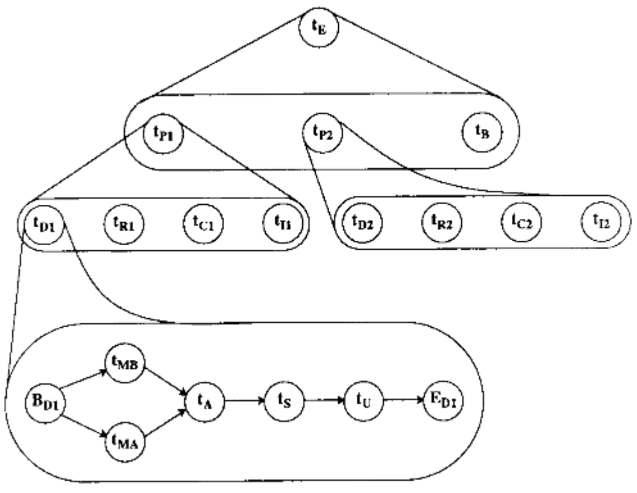

Fig. 2. Partial testing hierarchy for example circuit in Fig. 1.

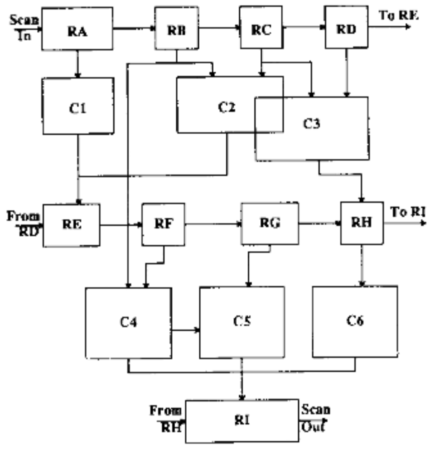

Fig. 3. An example system.

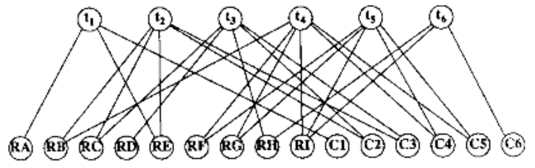

Fig. 4. Allocation graph A for the example system in Fig. 3.

III. Testability Resources and Test Modeling

A test t_i is defined in an unscheduled portion of the test hierarchy at which parallelism is to be investigated. For a typical BIST implementation, test t_i requires the use of a TPG, test control logic, the BUT, the intervening logic, and a signature analyzer (SA). Once such a *resource set* is known for each test, then it is possible to define an *allocation relation A* between tests and resources such that (t_i, r_j) is in A if resource r_j is in the resource set for test t_i.

The example given in Fig. 3 will be used to illustrate concepts in this and subsequent sections. The system shown consists of combinational logic denoted by C_i and registers denoted by R_i. Each combinational logic block is built-in testable and may represent testable PLA's, ROM's, or random blocks of logic. For this example, each of the registers is assumed to be configurable into one or more built-in logic block observer (BILBO) structures for testing attached combinational logic [3]. The BILBO's can be configured as a TPG or an SA as appropriate. A serial scan path is available for initializing TPG's and for observing the resulting signatures in the SA's. The use of this structure can be illustrated by test t_2 on block $C2$. Test t_2 uses RB and RC as a TPG and RE as an SA. In addition, block $C3$, because it overlaps block $C2$, is involved in the test. Thus, the resource set for test t_2 is $\{RB, RC, RE, C2, C3\}$, and (t_2, RB), (t_2, RC), (t_2, RE), $(t_2, C2)$, and $(t_2, C3)$ are all in relation A. In general, such information can be represented by a bipartite graph with a node set consisting of the tests and the resources. If (t_i, r_j) is in A, then there is an edge between t_i and r_j in the graph. Fig. 4 shows the *resource allocation graph* for the example system. In test t_5, it is assumed that the lateral signals from $C4$ to $C5$ are functionally dependent only on the inputs fed by register RF. It is also clear from Fig. 3 that t_4 must test both output paths to RI (from $C4$ and $C5$) separately.

In the resource allocation graph, a resource node which is connected to more than one test indicates contention between the tests for use of that resource. *In cases in which it is possible for a resource to be used simultaneously by two or more tests, the arcs between that resource node and the test nodes in the resource allocation graph are labeled with a common symbol.* Except for such labeling, it is assumed that tests sharing one or more resources must be disjoint in time, i.e., cannot be active concurrently. A pair of tests that cannot be run concurrently will be said to be *incompatible*. Otherwise, they are *compatible*.

Pairs of compatible tests form a relation on the set of tests which is a compatibility relation [6]. Such a relation can be represented by a *test compatibility graph* (TCG) in which a node appears for each test and an edge exists between node t_i and node t_j if test t_i and test t_j are compatible. The TCG for the

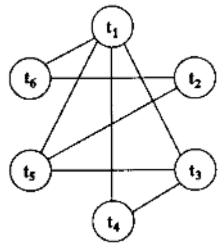

Fig. 5. Test compatibility graph TCG for the example system in Fig. 3.

example is shown in Fig. 5. The resulting TCG indicates exactly which pairs of tests can be run concurrently and from it one can derive larger sets of tests which also can be run concurrently if they exist.

The TCG can then be used as a basis for scheduling the tests so that the total testing time is minimized. In general, circuits fall into two classes 1) circuits in which all the tests are of approximately equal length, and 2) circuits in which the tests are unequal in length. Based on this classification, the following two problems may be stated: 1) find a schedule for running tests such that each test is run at least once and the total time to run all tests is minimum provided that each test t_i takes T units of time to run completely, and 2) find a schedule for running tests such that each test is run at least once and the total time to run all tests is minimum provided that a test t_i takes T_i units of time to run completely.

IV. Equal Length Tests

In [1], it is shown that for equal length tests, the total testing time for a particular schedule is nT where n is the number of test sets required to run all tests. A *concurrent test set CTS* is a set of tests which may be run concurrently. In order to obtain an optimal schedule (solution), the number of CTS's, n must be a minimum.

Theorem 1 [1]: Tests $t_{i_1}, t_{i_2}, \cdots, t_{i_p}$ can run concurrently if and only if $t_{i_1}, t_{i_2}, \cdots, t_{i_p}$ form a complete subgraph K_p in the TCG.

A *clique*, a maximal subgraph of a graph, of the TCG represents a maximal set of tests which can run concurrently. If all the cliques of the TCG were available, then finding S, a set of n cliques which covers all tests for n minimal, is an optimal solution to the equal length test scheduling problem.

Thus, the test scheduling problem reduces to 1) finding all the cliques of the TCG, and 2) solving the covering problem in order to determine a minimum collection. The following is the procedure as presented in [1].

1) Construct the TCG of the circuit.

2) Find G, the set of all cliques of the TCG. Let $G = \{G_1, G_2, \cdots, G_r\}$, where each G_i is a clique of the TCG.

3) By using a covering table, find a minimal subset S of G such that $\cup_S G_i = \{t_1, t_2, \cdots, t_q\}$, the set of all tests in the TCG.

4) Schedule all the tests in each G_i from S to run concurrently. The total testing time is $|S|*T$, where $|S|$ denotes the size of the set S.

The following example illustrates the steps of the procedure.

1) The TCG is given in Fig. 5.

2) Set G for the graph is $G = \{G_1, G_2, G_3, G_4, G_5\}$ in which $G_1 = \{t_1, t_3, t_5\}$, $G_2 = \{t_1, t_3, t_4\}$, $G_3 = \{t_1, t_6\}$, $G_4 = \{t_2, t_6\}$, $G_5 = \{t_2, t_5\}$.

3) The minimum cover can yield any one of the following three solutions: $S = \{G_2, G_1, G_4\}$, $S = \{G_2, G_4, G_5\}$, or $S = \{G_2, G_3, G_5\}$.

4) The total test time is $3T$.

The order in which the CTS's associated with different G_i's are run is not important. Also for tests which appear in more than one CTS in S, it is possible to delete all but one occurrence of such tests. The decision to eliminate redundant executions of a particular test is usually dependent on the test control implementation. It is also possible, however, to exploit these duplicate test executions to reduce the aliasing in the signature analyzers. For each execution of a test, a different configuration of the signature analyzer corresponding to a different polynomial can be used. This has been shown to reduce the probability of aliasing [7].

The minimum covering problem (step 3 of the optimal procedure) is NP-complete. In [8], the procedure from [1] was implemented using a suboptimal covering algorithm. Studies were performed for random test graphs. These studies showed that even for a reasonable number of tests the required computation became excessive. This was due primarily to the very large number of cliques generated in step 2, which produces enormous covering tables. These results prompted the need to find a heuristic which would generate a suboptimal solution without enumerating all of the cliques of the TCG. The approach taken was to develop an algorithm which generates exactly one complete set corresponding to each CTS in the final solution. Each CTS can also be thought of as an *independent set* of the complementary graph of the TCG, the *test incompatibility graph (TIG)*.

The problem of finding a minimum cover consisting of independent sets is equivalent to finding a minimum coloring of the TIG. The graph coloring problem tries to color each node in a graph such that no two adjacent nodes (nodes with a common incident edge) have the same color. A minimum coloring for a graph is one which requires a minimum number of colors [9]. The set of nodes having the same color in a minimum coloring is analogous to a CTS in a minimum equal length test schedule. This analogy is also reported in [10].

The approach of the algorithm is to generate a CTS by adding candidate tests to an expanding set. Once a CTS has been created, those nodes which are in the CTS are removed from the TIG and the remaining subgraph is used to generate a new CTS. This process is repeated until every test has been included (covered) in some CTS.

In a graph, the *degree* of a node (test) is the number of edges which are incident with that node (test). The algorithm selects a test t_i having maximum degree to seed a given expanding set (at each step the goal is to remove the maximum number of incompatibilities from the remaining graph). It is known that any node which is incompatible with test t_i cannot be a candidate to be added to the expanding set. Therefore, all adjacent (incompatible) nodes are added to a *marked set (M)* and excluded from further consideration during the expansion of the current CTS.

During the execution of the algorithm when no additional nodes can be added to the expanding set, a CTS has been found and M is the remaining subgraph of the TIG. To minimize the total number of CTS's in the final solution it is important, while processing the current expanding set, to minimize the incompatibilities in M. To achieve this goal, the algorithm first considers as candidates tests which are a distance 2 from the expanding set (tests compatible with the expanding set and incompatible to one or more of the tests in M). The set of distance 2 candidates is C'. If no such distance 2 candidates exist, then a test t_c is considered for inclusion in the expanding set if it is not an element of the marked set and has not yet been placed in the expanding set. The set C contains all candidate tests. Thus, a test t_i is selected from C' (or C if C' is empty) for inclusion in the expanding set such that its degree is maximum (removing the greatest number of incompatibilities in the remaining graph). The test(s) incompatible with test t_i are then added to M. The candidate sets are updated and the process continues until there are no new candidates.

For the formal presentation of the suboptimal algorithm, the following sets and notation are defined: S is the solution set, AN is the set of active (noncovered) nodes, T is the set of all nodes, M is the set of "marked" nodes, C is the set of candidate nodes, C' is the set of distance 2 candidate nodes, $N(\{t_i\})$ is the adjacency relation (set of nodes incompatible with a set of nodes $\{t_i\}$), $|t_i|$ is the degree of node t_i. Algorithm 1 is the resulting suboptimal algorithm.

Algorithm 1: suboptimal algorithm for equal length tests.

```
$k \leftarrow 0$; $AN \leftarrow T$; $S \leftarrow \emptyset$;
while $AN \neq \emptyset$
begin
    $k \leftarrow k+1$; $M \leftarrow \emptyset$; $C \leftarrow AN$; $C' \leftarrow \emptyset$; $S_k \leftarrow \emptyset$;
    while $AN \cap \bar{M} \neq \emptyset$
    begin
        if $C' \neq \emptyset$, choose $t_i \in C'$ where $|t_i|$ is maximum,
        else choose $t_i \in C$ where $|t_i|$ is maximum;
        $AN \leftarrow AN - \{t_i\}$; $M \leftarrow M \cup (N(\{t_i\}) \cap AN)$; $C \leftarrow C -$
        $(\{t_i\} \cup N(\{t_i\}))$;
        $C' \leftarrow C \cap N(M)$; $S_k \leftarrow S_k \cup \{t_i\}$;
    end
    $S \leftarrow S \cup \{S_k\}$;
end.
```

In the implementation of the algorithm, if there is more than one t_i such that $|t_i|$ is maximum, the test with smallest subscript is chosen.

The algorithm generates the following solution for the complementary graph of Fig. 5.

$$S = \{S_1 = \{t_2, t_6\},\ S_2 = \{t_1, t_3, t_4\},\ S_3 = \{t_5\}\}.$$

In this example the solution is optimal and S_1 and S_2 correspond to G_4 and G_2, respectively. Note that in general, the solution CTS's $S_1, S_2, \cdots, S_j$ will not be cliques. This heuristic produces an irredundant solution (each test appears exactly once in the final solution). If cliques are desired, each set can be expanded to generate cliques for the solution by adding to each set all tests which are compatible with every test in the set. As an example, S_3 can be expanded to correspond to either G_1 or G_5.

TABLE I
RESULTS FOR NINE RANDOM GRAPHS

# of Tests	Enumeration of Cliques & Suboptimal Cover				Algorithm 1	
	# of Cliques	Clique Generation Time	# of Test Sets	Total Time	# of Test Sets	Execution Time
30	978	4.3	5*	48.8	5*	0.2
30	844	4.0	5	36.0	4*	0.2
30	930	3.7	4*	40.1	4*	0.2
40	5161	29.1	5	944.0	5	0.4
40	4564	20.0	8*	822.6	8*	0.4
40	4931	26.7	6*	825.2	6*	0.4
50	20119	117.9	10*	14445.7	10*	0.7
50	17978	125.4	7	10803.2	6	0.6
50	20474	109.6	7*	13496.3	7*	0.6

* Indicates the solution is optimal. All times are CPU seconds on a VAX 11/750.

The selection of a test t_i and the updating of all sets is executed once for each test. The determination of C' is on the order $O(N^2)$ (N is the number of tests). Therefore, the complexity of the algorithm is on the order $O(N^3)$.

Algorithm 1, although developed independently, appears to utilize the same heuristics as the suboptimal graph coloring algorithm BSP2 reported in [11]. In this survey [11], several suboptimal algorithms for graph coloring are compared and BSP2 is reported to be superior for large graphs.

Table I shows the results of performance measurements made on the two implementations of the equal length test problem using a series of randomly generated graphs. The results for the implementation of procedure from [1] using a suboptimal covering algorithm as reported in [8] are listed in the left portion of Table I. Execution times are reported for both clique generation and the combined clique generation/clique cover time. It is important to note how rapidly the number of cliques increases as the number of tests (nodes) in the graph increases.

The right portion of the table presents the results for Algorithm 1. Algorithm 1 produced optimal or near-optimal solutions in all cases. Even for large TCG's, the execution times are very reasonable. For all of the random graphs, a lower bound for the optimal number of test sets was calculated. This lower bound is the size of the largest clique of the TIG. The solutions are marked with an asterisk and are optimal if the number of CTS's in the solution matches the lower bound figure for a particular graph. It should be noted that the fact that some of the results differed from the lower bound does not indicate that the solutions were not optimal. For the purposes of this work, however, since the solutions were adequately "close" to the lower bound, the excessive cost of determining true optimal solutions for comparison was not justified.

Another study on 60 random TIG's, ranging in size from 30 to 50 nodes, resulted in the suboptimal algorithm producing an optimal solution for 34 TIG's. In 24 other TIG's, the algorithm produced a solution which contained one additional CTS than was indicated by the lower bound estimate of optimal. In the remaining two graphs, the algorithm strayed by two CTS's (all graphs produced solutions ranging from 3 to 10 CTS's).

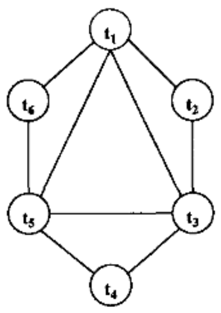

Fig. 6. TCG for unequal length test example.

V. Unequal Length Tests

In the unequal length test problem, a test t_i requires T_i units of time to run completely. Consider the added constraint associated with the unequal length test problem where two tests t_1 and t_2 can run concurrently and $T_1 > T_2$. Clearly, if both t_1 and t_2 are initiated simultaneously, t_2 will finish before t_1. Three possibilities now exist. 1) Test t_2 can be modified to take T_1 units of time to run completely or alternatively t_2 can be stopped and its results saved. Thus, only on completion of t_1 can the analysis of both t_1 and t_2 be performed internally by a local controller or externally after the results have been accessed. 2) If local test analysis hardware exists internally, then local control can be provided to process both t_1 and t_2 independently. Therefore, in such an environment, t_1 proceeds uninterrupted and a new test, say t_3, which is compatible with t_1(t_3 incompatible with t_2) can be initiated. However, it is important to note that each test once initiated must run to completion. 3) A third possibility exists if a mechanism which permits storing and restoring of test state is available. Test t_1 is interrupted upon completion of test t_2, the results of t_2 along with the status of t_1 and also the partial results due to test t_1 are saved. The results of t_2 can be compared and a new set of tests can be started. If the new set includes t_1, then t_1 need only be restarted from its interrupted state after restoring the status of t_1 and its partial results. Alternatively, the unfinished segment of t_1 can be completed at a later time.

The three possibilities will be explained by way of an example. Consider the TCG shown in Fig. 6. The time for the completion of each of the tests is $T_1 = T_3 = T_5 = 2T$, $T_2 = T_4 = T_6 = T$. In the first case, shown in the Fig. 7(a), tests t_1, t_2, t_3 are initiated simultaneously. Test t_2, being of shorter duration than t_1 and t_3, is adjusted to run for a duration of $2T$. Thus, at time $2T$ the results of all three tests are evaluated. At this time, tests t_4 and t_5 are initiated and test t_4 is extended to run for the duration of $2T$ units. Thus, the results of the tests t_4 and t_5 are evaluated at time $4T$ and test t_6 is initiated at that time. Total testing time is five units. This is an optimal schedule under the condition that each test once started must run to completion and no test may be interrupted. This scheduling discipline, which will be referred to as *nonpartitioned testing*, was discussed in [1].

In the second case, tests t_1, t_2, and t_3 are initiated as is evident from Fig. 7(b). On completion of t_2, a new test t_5 is scheduled while test t_1 and t_3 are still running. Test t_4 is

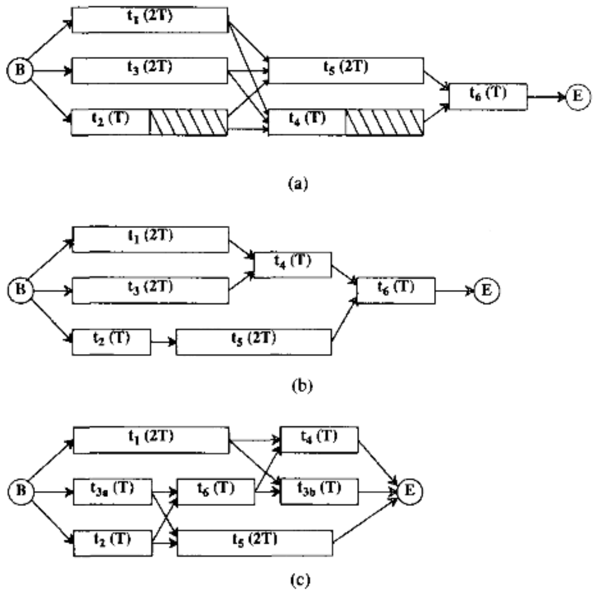

Fig. 7. Time schedules for unequal length tests: (a) nonpartitioned testing, (b) partitioned testing with run to completion, and (c) partitioned testing.

scheduled on completion of t_1 and t_3 and finally, test t_6 is scheduled last. Total testing time is four units. This is an optimal schedule under the condition that each test once started must run to completion. Note that in a serial scan environment, if signatures are compared externally, this scheme may imply interrupting t_1 and t_3 on completion of t_2, then restarting t_1 and t_3 from interrupted states. This scheduling discipline will be referred to as *partitioned testing with run to completion*.

In the third case, each test can be interrupted at any time, the only requirement being that all tests must be completed by the end of testing. As shown in Fig. 7(c), test t_3 in this case is run in two segments. Test t_3 is initiated at time 0. At time T, when all tests are interrupted, test t_3 is not rescheduled; instead its status and partial results are saved and t_3 is rescheduled at time $2T$ after restoring its saved state and partial results. Total testing time in this case is three units, which is once again optimal under the condition that tests can be interrupted and restarted at will. This scheduling discipline will be referred to as *partitioned testing*.

In the above discussion, it has been assumed that the time spent in saving and restoring of the partial state and results for a test and for the comparison of results is negligibly small compared to the duration of any segment of the test. If the time overhead due to interruption is not small, then such time should also be added to the total time for testing.

An algorithm was presented in [1] to find an optimal schedule for the case of nonpartitioned testing. It was shown that not only is it necessary to find all cliques, but also subcliques of the cliques before solving the covering problem to find a minimum cover. Thus, the computational complexity of this problem is considerably greater than the equal length algorithm. Attempts were made to develop suboptimal algorithms to schedule unequal length tests in the nonpartitioned testing scheduling discipline. These efforts met with little success.

However, success of the equal length heuristic, presented in Section IV of this paper, prompted the use of the same heuristic for the unequal length problem. In what follows, partitioned testing is reduced to an equal length test problem. It is then suggested that the equal length heuristic be applied to the different cases of unequal length problem. It is important to keep in mind that if a test t_i is segmented into $\{t_{i_1}, t_{i_2}, \cdots, t_{i_{m_i}}\}$, then this collection of segments frequently needs to be treated as an ordered set for the scheduling of segments during testing. In other words, in the final schedule, not only must all segments appear, but all t_{i_j} must appear before t_{i_k} for all $1 \leq j < k \leq m_i$. The need for ordering is illustrated by the case in which the signature is determined over the entire sequence of test segments for a given test. It should be noted that any sequential ordering of test vectors otherwise required must also be preserved in the segmenting process.

Theorem 2: Partitioned testing for the unequal length test problem can be reduced to an equivalent equal length test problem.

Proof: The assertion is proved in two parts. In part I, the TCG for the unequal length test case is redefined such that all the nodes in the new graph correspond to tests of equal length. In part II, it is shown that an optimal schedule derived for the modified graph can be used to derive an optimal schedule for partitioned testing for the unequal length test problem.

Part I: Let the original TCG for the system in which tests are of unequal lengths be TCG_u. Clearly in TCG_u every node corresponds to a test t_i of length T_i. Let there be m nodes in the graph. Let

$$T = \gcd\{T_1, T_2, \cdots, T_m\}.$$

Note that T divides T_i for all $1 \leq i \leq m$. Let $a_i = T_i/T$. Expand TCG_u to a new graph TCG_e as follows: for every node t_i in TCG_u there are a_i nodes $t_{i_1}, t_{i_2}, \cdots, t_{i_{a_i}}$ in TCG_e. All the a_i nodes corresponding to the node t_i are incompatible with each other. Two nodes t_{i_j} and t_{k_l} in TCG_e are compatible if and only if t_i and t_k are compatible in TCG_u. It is easy to see that TCG_e so obtained from TCG_u is a much larger graph with the total number of nodes being $\Sigma_{i=1}^{m} a_i$. Furthermore, in TCG_e each node corresponds to a test of length T, although there is no information on the order of segments of a test t_i in TCG_e.

Part II: Let $S = \{G_1, G_2, \cdots, G_k\}$ be a minimum cover for some TCG_e. In the case of the equal length test problem, it was pointed out that the total time for testing is independent of the order in which tests are applied. However, in the case of unequal length tests, it is important that an interrupted test be restarted from its preinterrupt state. Thus, if t_i was interrupted at t_{i_j}, then whenever t_i is restarted it must start from $t_{i_{j+1}}$. It will now be shown that this can be achieved once a minimum cover is found for the graph TCG_e. Without loss of generality, assume that the test schedule is $G_1\ G_2 \cdots G_k$, i.e., all tests of G_1 are applied first and they are followed by all tests of G_2, etc. Furthermore, assume that every test t_{i_j} appears only once in this schedule. In this schedule, let Gj_1, $Gj_2, \cdots, Gj_{a_j}$ be all those G's which contain segments of a test t_j. Note that no set Gj_k can contain more than one segment of the test t_j because all such segments are incompatible with each other. We now modify Gj_k such that t_{j_k} replaces the segment of t_j in Gj_k for all $1 \leq k \leq a_j$. This process now assures that in a given schedule, segments of the test t_j always appear in the desired order. This process is repeated for every test t_i, $1 \leq i \leq m$. □

Suboptimal algorithms were implemented to solve the partitioned testing with run to completion and partitioned testing cases of the unequal length test scheduling problem. Both algorithms convert the unequal length TCG_u into an equivalent equal length TCG_e. Rather than actually produce the TCG_e, however, it is possible to take advantage of the fact that each unequal length test maps to a set of equal length segments each having an identical compatibility relation. The algorithm models each set of equal length segments as a set of tokens via two parameters: 1) the number of remaining equal length segments, and 2) the node degree of the next segment to be selected. Thus, except for additional bookkeeping, the partitioned testing problem can be solved using the equal length algorithm. It should also be noted that by processing the set of equal length segments in this manner, the proper ordering of the segments in the final solution can easily be maintained.

The details of the partitioned testing algorithm are presented in Algorithm 2 with the following sets and notation. Sets S, AN, T, M, C, and C', relation $N(\{t_i\})$, and degree operator $|t_i|$ are equivalent to the sets and operators described for Algorithm 1. In addition, L_i is the number of equal length segments for test t_i, and I_i is the number of equivalent incompatibilities (degree) associated with each token for t_i.

Algorithm 2: suboptimal algorithm for partitioned testing.

```
$k \leftarrow 0$; $AN \leftarrow T$; $S \leftarrow \emptyset$; $L_i \leftarrow a_i$; $I_i \leftarrow L_i - 1 + \Sigma_{j | t_j \in N(\{t_i\})} L_j$;
while $AN \neq \emptyset$
begin
  $k \leftarrow k+1$; $M \leftarrow \emptyset$; $C \leftarrow AN$; $C' \leftarrow \emptyset$; $S_k \leftarrow \emptyset$;
  while $AN \cap \bar{M} \neq \emptyset$
  begin
    if $C' \neq \emptyset$, choose $t_i \in C'$ where $I_i$ is maximum.
    else choose $t_i \in C$ where $I_i$ is maximum;
    $AN \leftarrow AN - \{t_i\}$; $M \leftarrow M \cup (N(\{t_i\}) \cap AN)$; $C \leftarrow C$
    $- (\{t_i) \cup N(\{t_i\}))$;
    $C' \leftarrow C \cap N(M)$; $S_k \leftarrow S_k \cup \{t_i\}$; $I_i \leftarrow I_i - 1$; $I_j \leftarrow I_j - 1$
    where $(j \in N(\{t_i\}))$;
  end
  $S \leftarrow S \cup \{S_k\}$;
  for each $t_i \in S_k$
  begin
    $L_i \leftarrow L_i - 1$;
    if $L_i \neq 0$
      then $AN \leftarrow AN \cup \{t_i\}$;
  end
end.
```

Algorithm 2 was modified for partitioned testing with run to completion (Algorithm 2M). In Algorithm 2M, an initial CTS is generated. For the partitioned testing with run to completion, it is required that all segments of a given test be run

TABLE II
RESULTS FOR ALGORITHM 2 AND ALGORITHM 2M

Number of nodes in unequal length graphs	Number of graphs	Mean number of equivalent equal length nodes	Mean percent over lower bound for Algorithm 2	Worst case percent over lower bound for Algorithm 2	Mean percent over lower bound for Algorithm 2M
20	7	2163	.3	2.1	2.0
30	27	2168	.8	9.6	2.0
40	16	3271	1.0	14.1	2.8
50	14	5085	1.1	5.9	2.3

contiguously. Thus, all of the tests in the CTS must be scheduled to run in parallel until the shortest currently active test t_k is completed. At this point, the current CTS S is modified to reflect the removal of all completed tests and this modified set is used as a seed for further processing in order to preserve the run to completion requirement. This generates a test schedule of the type shown in Fig. 7(b).

Algorithm 2M: suboptimal algorithm for partitioned test with run to completion.

```
$AN \leftarrow T$; $S \leftarrow \emptyset$; $L_i \leftarrow a_i$; $I_i \leftarrow L_i - 1 + \Sigma_{j|t_j \in N(\{t_i\})} L_j$;
while $AN \neq \emptyset$
begin
  $M \leftarrow N(S)$; $C \leftarrow AN \cap \bar{M}$; $C' \leftarrow C \cap N(M)$;
  while $C \neq \emptyset$
  begin
    if $C' \neq \emptyset$, choose $t_i \in C'$ where $I_i$ is maximum,
    else choose $t_i \in C$ where $I_i$ is maximum;
    $AN \leftarrow AN - \{t_i\}$; $M \leftarrow M \cup (N(\{t_i\}) \cap AN)$; $C \leftarrow C$
    $- (\{t_i\} \cup N(\{t_i\}))$;
    $C' \leftarrow C \cap N(M)$; $S \leftarrow S \cup \{t_i\}$; $I_i \leftarrow I_i - 1$; $I_j \leftarrow I_j - 1$
    where $(j \in N(\{t_i\}))$;
  end
  time$\leftarrow \min_{i|t_i \in S} \{L_i\}$;
    {output $S$ and time; current solution $S$ to run for
    time$*T$ units}
  for all $t_i \in S$
  begin
    $L_i \leftarrow L_i -$ time; $I_i \leftarrow I_i -$ (time $- 1$);
    for $t_j \in N(\{t_i\})$
    begin
      $I_j \leftarrow I_j -$ (time $- 1$);
    end
    if $L_i = 0$ then $S \leftarrow S - \{t_i\}$;
  end
end.
```

Table II shows the comparative results for the two suboptimal unequal length algorithms. The lower bound estimate calculated for each graph is the test time determined by the largest weighted clique of the TIG. It should be noted that this is a very loose lower bound approximation. The table shows that for the majority of the graphs, both algorithms perform very well. The worst case CPU time for Algorithm 2 was approximately 15 s on a VAX 11/750 for an equivalent equal length graph size of 7000 nodes.

As a final illustration of the unequal length test schedule, consider the TIG (Fig. 8) for the combined second and third levels of the example model shown in Fig. 2. The time for the completion of each of the tests is $T_{D1} = T_{D2} = 20$, $T_{R1} =$

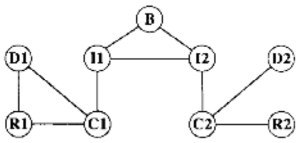

Fig. 8. TIG for testing the combined second and third levels of example.

$T_{R2} = 6$, $T_{C1} = T_{C2} = 10$, and $T_{I1} = T_{I2} = T_B = 2$. The optimal solution is bounded by the clique comprised of nodes $\{D1, R1, C1\}$, which implies a minimum execution time of 36 units. One solution for the nonpartitioned testing is $G_1 = \{D1, D2, R2, I1\}$, $G_2 = \{C1, C2, B\}$, and $G_3 = \{R1, I2\}$. This solution yields a total testing time of 36 units. The optimal solution for partitioned testing with run to completion and partitioned testing is also 36 units.

VI. CONTROL

Having determined a schedule, the problem becomes one of implementing the test schedule in a cost-effective way. This requires a low-cost solution to the problem of controlling different resources used in the application of different tests. Resources used by tests in a CTS must be initialized and the test results must be observed at appropriate instants by a supervisor. Such a supervisor or controller can be centralized or distributed, but it must meet the following objectives. 1) It must initialize the required resources for different tests at appropriate instants. 2) It must control all resources used in performing a test including the intervening logic for the application of appropriate tests. 3) The supervisor must be able to communicate with BUT's, aggregate the test results if required, and communicate with the external world. (Depending on the choice of implementation, the supervisor may be expected to communicate with other supervisors or, in the case of hierarchical design, it should be able to communicate with higher level supervisors.) 4) The complete control structure should be such that the supervisor is simple and area efficient and the communication is reliable with low overhead (in space as well as in time).

In this section, minimal self-testing units necessary for the application of tests which place resource control physically close to the BUT are described; then a number of structures to implement the control are discussed.

For the testing control, a *self-testing unit (STU)* is defined to be comprised of a BUT and its corresponding *test control logic (TCL)*. For this discussion, it is assumed that each BUT is sufficiently complex and has enough unique features to warrant a local control of its own. An STU is conceptually similar to an SV as defined in [12]. The structure of an STU is such that the resources required to test a BUT are controlled and initialized by the TCL associated with the BUT. Such resources can be external to a TCL or alternatively a TCL may contain BIST elements (linear feedback shift register, counter, multiplexer, etc.) required for testing its associated BUT. Also, in certain structures it may not be possible to draw a distinction or boundaries between the TCL and BUT. For example, in BIST PLA's [13], [14], extra logic used during testing is interspersed in the PLA. Similarly, in the case of a

TABLE III
A QUALITATIVE COMPARISON OF THE THREE MAIN CONTROL STRUCTURES

Control Structure	Communication Area Requirements	TCL Complexity	System Reliability	Testing Time Effects	Most Appropriate Scheduling Discipline
Star	High	Low	High	None	Partitioned testing
Bus	Low	High	Low	Increased	Non-partitioned testing
Multiple Bus	Moderate	High	Moderate	Moderately Increased	Partitioned testing with run to completion

BUT being a microprogrammed ALU, the TCL may be implemented as part of the microprogram to test the ALU.

It is quite likely that two tests, t_1 and t_2, share common test resources (note such tests are normally incompatible). In such an environment, a resource common between two distinct blocks must be controllable by two TCL's. It is important, in such cases, that TCL's are activated such that no conflicts result in the use of resources.

Having defined STU's as above, the function of the supervisor is essentially to control the STU's and communicate with them and also with the external world. Of course, the supervisor must control STU's such that no conflict arises in resource utilization. In other words, the supervisor must implement the solution to the test scheduling problem. Briefly, the functions of a supervisor are as follows: 1) send control signals (initialize, start testing, result request) to STU's, 2) obtain test results from STU's, 3) collect test results and analyze the state of STU's, and 4) communicate the outcome of testing to higher level supervisors or the external world. Note, such information often may need to be communicated in coded form to save on pin count [15].

The type of implementation used for both the communication between a supervisor and its STU's and between a supervisor and the external world will be referred to as the control structure. Three potential control structures are considered: 1) a star structure, 2) a bus structure, and 3) a multiple bus structure. Comparative results on these three structures are given in Table III. These results will be briefly discussed here for each structure.

Star Structure: The simplest and most general method to implement the control is by a star configuration in which the supervisor is directly connected to each STU. In this approach, the supervisor can control/communicate with each STU independently. Clearly, if k_i is the number of communication lines between the supervisor and the ith STU, then the total number of communication lines will be $\Sigma_{i=1}^{n} k_i$, for a system with n STU's. Because of the dedicated nature of the communication paths in the star, the communication is comparatively of higher reliability since information can still be collected from TCL's connected to other than the faulty line.

Since the star is the most general control scheme, it can be used to implement any one of the three scheduling disciplines discussed in Section V. It is ideally suited for implementing partitioned testing because any test can be interrupted at any time by the supervisor without any effect on the other tests. Thus, the overhead due to the interrupts can be minimized.

A minor generalization of the star is a multilevel tree in which there are one or more intermediate levels of supervisors between the primary supervisor and the STU's. Although the number of communication lines increases in this case, a saving in area can take place by locating the lower level supervisors physically close to the STU's they control. One of the advantages of this scheme is that it can easily be extended to board or system level test.

Bus Structure: The number of communication lines and, therefore, area occupied by a communications network can be reduced by using a bus structure for supervisor–STU communication. In this scheme, each STU is assigned an identification, ID. The supervisor can communicate with STU's through their ID's. Although the number of communication lines decreases, the complexity of TCL's increases as each TCL must be able to identify its ID.

In the simplest environment, a supervisor can start one test at a time through a TCL but at any given instant many tests can be running concurrently. Similarly, for the communication of results, only one TCL can send results to a supervisor. To avoid any collisions [16] on the bus, TCL's can be required to send results to a supervisor only on receipt of a result request signal from the supervisor. This may increase the total testing time as the results will be transmitted sequentially. Alternatively, codes can be employed in which a number of TCL's can transmit results simultaneously on a single bus [17], [18] without unduly effecting the reliability of transmission while reducing the time spent in transmission of results. Any fault within the bus structure may disable the entire structure, thus cutting off communication between the supervisor and all TCL's; thus, the reliability of this structure is comparatively low.

In a bus-oriented control structure, a considerable time will be spent in initialization of tests and result communication; therefore, this control scheme is best suited for nonpartitioned testing where initialization and result communication are performed relatively infrequently.

Although the use of such a scheme may raise other questions (e.g., synchronization etc.), nontheless it makes the point that parallelism may be achievable even in bus-oriented control structure. A factor which provides support to the use of a bus-oriented structure is that in some VLSI architectures it may be possible to use existing on-chip buses during test mode. A major drawback of this scheme is that the bus becomes an essential testing structure. Thus, if the bus fails, no testing may proceed and very little diagnosis information may be obtained.

Multiple Bus Structure: The bus and star structures proposed above can be combined to include more than one bus. Two alternative structures and allocation of STU's to buses for the multiple bus structure are given in [18]. One such configuration would be to have k buses (k equals the maximum number of tests in a CTS). The tests which are simultaneously interrupted are assigned to different buses. This structure is particularly ideal for use with the partitioned test with run to completion scheduling discipline because the partitioning of STU's to different buses can eliminate sequentially processed interrupts. Generally, in the multiple bus structure, buses can be organized based on the physical location of STU's. Thus, the choice of the structure which minimizes the overhead can be made. It should be noted that the multiple bus structure can be generalized to a hierarchical structure by including a higher level supervisor to control lower level supervisors with supervisor communication implemented in either a star or multiple bus structure.

In the above discussion, the approach has been to implement the control in a hierarchical manner. Although the existence of a centralized controller has been assumed, test resources have been assumed to be controllable by TCL's. Alternatively, it is possible to control all resources by a single centralized controller as proposed in [17]. It is the opinion of the authors that the use of TCL's which are placed physically close to the resources they control is a more flexible implementation. In addition, the use of TCL's reduces the complexity of the central controller and it offers the freedom to use different control structures which in turn can reduce the area and/or time overhead.

Yet another alternative is to distribute part of the control. In one of the simplest schemes, TCL's can be issued tokens [16] at the start of the test cycle. Upon completion of a test, the tokens are passed on to other TCL's which in turn can start testing their BUT's. This process continues until the complete system has been tested.

VII. Conclusions

In this paper, the test scheduling problem for equal length and unequal length tests for VLSI circuits using BIST has been modeled. An efficient algorithm has been presented for producing test schedules for the equal length tests. The unequal length test problem has been subdivided into three cases and efficient algorithms have been presented for two out of the three cases. Although the solutions presented here have been given in the context of BIST, it is also possible to employ the results in other testing environments where inherent parallelism exists. In addition, the solutions presented here for VLSI circuits using BIST can be extended to the board or system level as well.

The algorithms presented for test scheduling often yield optimum solutions. In the most extreme departure from optimum among the test cases, the excess length of a test schedule was no more than 15 percent. It is thought that for most testing applications, this degree of optimization is likely to be adequate. It should be noted that the algorithms presented may have applications beyond testing since they can be applied to any situation in which an efficient fixed schedule is to be defined for a fixed allocation of tasks to resources.

Finally, a very general test control model has been introduced for use in the BIST environment. Several control structures and their appropriateness for use with various scheduling cases defined is discussed.

References

[1] C. Kime and K. Saluja, "Test scheduling in testable VLSI circuits," in *Proc. Int. Symp. Fault-Tolerant Comput.*, Santa Monica, CA, June 1982, pp. 406–412.
[2] *IEEE Design and Test of Computers*, vol. 2, Apr. 1985.
[3] B. Konemann, J. Mucha, and G. Zwielhoff, "Built-in logic block observation techniques," in *Proc. Int. Test Conf.*, Cherry Hill, NJ, Oct. 1979, pp. 37–44.
[4] M. Abadir and M. Breuer, "Constructing optimal test schedules for VLSI circuits having built-in test hardware," in *Proc. Int. Symp. Fault-Tolerant Comput.*, Ann Arbor, MI, June 1985, pp. 165-170.
[5] R. Grimaldi, *Discrete and Combinatorial Mathematics—An Applied Introduction*. Reading, MA: Addison-Wesley, 1985, pp. 141–145.
[6] Z. Kohavi, *Switching and Finite Automata Theory*, 2nd ed. New York: McGraw-Hill, 1978, pp. 28, 333-347.
[7] S. Hassan and E. McCluskey, "Increased fault coverage through multiple signatures," in *Proc. Int. Symp. Fault-Tolerant Comput.*, Orlando, FL, June 1984, pp. 354–359.
[8] G. Craig and C. Kime, "Determining parallel test schedules for VLSI built-in test," Tech. Rep. ECE-84-23, Dep. Elec. Comput. Eng., Univ. Wisconsin-Madison, Sept. 1984.
[9] M. Swamy and K. Thulasiraman, *Graphs, Networks, and Algorithms*. New York: Wiley, 1981.
[10] A. Krasniewski and A. Albicki, "Automatic design of exhaustively self-testing chips with BILBO modules," in *Proc. Int. Test Conf.*, Philadelphia, PA, Nov. 1985, pp. 362–371.
[11] A. Shneider, "Classification analysis of heuristic algorithms for graph coloring," *Cybernetics*, vol. 20, pp. 484–492, 1984.
[12] R. Sedmark, "Implementation techniques for self-verification," in *Proc. Int. Test Conf.*, Cherry Hill, NJ, Oct. 1980, pp. 267–278.
[13] R. Treuer, H. Fujiwara, and V. Agarwal, "Implementing a built-in self-test PLA," *IEEE Design Test*, vol. 2, pp. 37–48, Apr. 1985.
[14] K. Saluja and J. Upadhyaya, "A built-in self-test PLA design with extremely high fault coverage," in *Proc. IEEE Int. Conf. Comput. Design*, Port Chester, NY, Oct. 1986, pp. 596–599.
[15] R. Sedmak and H. Liebergot, "Fault-tolerance of a general purpose computer implemented by very large scale integration," *IEEE Trans. Comput.*, vol. C-29, pp. 492–500, June 1980.
[16] A. Tanenbaum, *Computer Networks*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
[17] J. Beausang and A. Albicki, "Towards determining an optimal test control line distribution scheme for a self-testable chip," Tech. Rep. EL-86-04, Dep. Elec. Eng., Univ. Rochester, Rochester, NY, Mar. 1986.
[18] K. Saluja, C. Kime, and G. Craig, "Design of control for scheduling tests in testable VLSI circuits," Tech. Rep. EE8540, Dep. Elec. Comput. Eng., Univ. Newcastle, New South Wales, Australia, 1985.

Gary L. Craig (S'79–M'87) received the B.S. degree in electrical engineering from West Virginia University, Morgantown, in 1982 and the M.S. and Ph.D. degrees in electrical engineering from the University of Wisconsin, Madison, in 1984 and 1987, respectively.

He is presently an Assistant Professor in the Department of Electrical and Computer Engineering, Syracuse University, Syracuse, NY. His current research and teaching interests are in design for testability, VLSI built-in self-test, VLSI design, fault-tolerant computing, and computer architecture.

Charles R. Kime (S'66–M'66–SM'79) received the B.S. degree from the University of Iowa, Iowa City, in 1962, the M.S. degree from the University of Illinois in 1963, and the Ph.D. degree from the University of Iowa in 1966, all in electrical engineering.

He joined the faculty of the University of Wisconsin, Madison, in 1966 where he is currently a Professor in the Department of Electrical and Computer Engineering. He spent the 1973–1974 academic year as a Visiting Associate Professor in the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. His current research and teaching interests are in testing and built-in test, VLSI systems design, fault-tolerant computing, computer architecture, and logic design.

Dr. Kime is a member of the Association for Computing Machinery, the American Society of Engineering Education, and Sigma Xi. He has served as an Associate Editor of the IEEE TRANSACTIONS ON COMPUTERS and as the General Chairman of the 1979 International Symposium on Fault-Tolerant Computing.

Kewal K. Saluja (S'70–M'73) received the B.E. (Elect.) degree from the University of Roorkee, Roorkee, India, in 1967 and the M.S. and Ph.D. degrees in electrical engineering from the University of Iowa, Iowa City, in 1972 and 1973, respectively.

From 1967 to 1970 he was with the state electricity board, U.P. India. From 1973 to 1985 he was with the Department of Electrical and Computer Engineering, University of Newcastle, N.S.W., Australia. He is presently an Associate Professor in the Department of Electrical and Computer Engineering at the University of Wisconsin, Madison. He has held visiting and consulting positions at the University of Southern California, the University of Wisconsin, the University of Iowa, the State University of New York, Binghamton, Hiroshima University, and other institutions. His research interests include design for testability, fault-tolerant computing, VLSI design, and computer architecture. His teaching interests are logic design, mini and microcomputer system architecture, and VLSI design and testing.