Efficient Path Delay Test Generation for Custom Designs

Sungho Kang, Bill Underwood, Wai-On Law, and Haluk Konuk

Due to the rapidly growing complexity of VLSI circuits, test methodologies based on delay testing become popular. However, most approaches cannot handle custom logic blocks which are described by logic functions rather than by circuit primitive elements. To overcome this problem, a new path delay test generation algorithm is developed for custom designs. The results using benchmark circuits and real designs prove the efficiency of the new algorithm. The new test generation algorithm can be applied to designs employing intellectual property (IP) circuits whose implementation details are either unknown or unavailable.

Manuscript received February 17, 2000; revised August 1, 2001.

Sungho Kang (Phone: +82 2 2123 2775, e-mail: shkang@yonsei.ac.kr) is with the Yonsei University, Seoul, Korea.

Bill Underwood (e-mail: billu@synopsys.com) is with the Synopsys Inc., Austin, USA.

Wai-On Law (e-mail: wai-on.law@motorola.com) is with the Motorola Inc., Hong Kong.

Haluk Konuk is with the Agilent Technologies Inc., USA.

I. INTRODUCTION

Delay testing refers to any test whose objective is to assure that a circuit operates correctly at a given clock rate. Importance of delay testing is growing especially for high speed circuits. There are two widely used fault models for delay testing. The gate delay model assumes that all the excessive delay is localized in a single faulty gate. This model has the attractive property that one needs to add to the stuck-at fault test vector only a preliminary vector setting the opposite logic value on the gate under test in order to convert the stuck-at fault test vector into a vector to test for the fault. Unfortunately, the gate delay fault model fails to consider certain types of potential delay defects, such as distributed defects that consist of small delays on each gate in the circuit. The path delay model, however, assumes that a circuit fails to operate at its specified clock frequency because of just such small delays that accumulate along the entirety of a path being tested. A test set based on this model will detect both localized and distributed delay defects along circuit paths. Unfortunately, there are a very great number of paths in large circuits, so that testing of all path delay faults may be prohibitively expensive.

However, the advantage of being able to detect small distributed delays provides a mechanism for monitoring process variations that can have a subtle but deleterious effect on paths that are near the timing limit for the circuit. Since a standard stuck-at fault test set was to be used and would be applied at full clock speed, it was assumed that localized delays would be tested adequately by the stuck-at fault tests and gate delay tests. Therefore, the further capability needed is the ability to test for path delays. Since a test set for all delay paths is infeasible, tests are to be generated only for the longest functional paths in the design as indicated by a static timing analysis tool. A process

called binning is often used to sort microprocessors into different versions depending on the clock speed at which they can be guaranteed to run. Tests for path delay faults on the longest paths in a circuit provide an ideal vehicle for binning, since they give the most accurate picture of the clock speed at which failures begin to occur.

There are multiple types of tests that can be generated for path delay faults according to the detection capabilities. A hazard-free robust (HFR) test for a path delay fault is a two-vector test that guarantees that the signals on the path are free from dynamic hazards and that guarantees to detect the delay fault independent of the delays in the rest of the circuit. This type of test is ideal for delay fault diagnosis, since the test will fail if and only if the path under test has excessive delay [1]. A second type of test is the robust (ROB) test, which guarantees to detect the delay fault independent of the delays in the rest of the circuit but does not guarantee that signals on the path are free from dynamic hazards. Certain types of test for a path delay fault that do not meet the requirements of a robust test are called non-robust tests. While this may imply that all non-robust tests are equally good, however, we wish to distinguish among several sub-types of non-robust tests. One type of non-robust test is a strong non-robust (SNR) test [2]. A strong non-robust test requires the same logic values in each of the two clock cycles as does a robust test, but those off-path inputs that are required to maintain the same logic value for both clock cycles may have static hazards. The other type is a weak non-robust (WNR) test which is often called as a non-robust test [3]. This requires that the values in the second clock cycle be the same as those for the robust test, while the only required value for the first clock cycle is the initial value at the head of the path.

Most of researches in delay testing focus on circuits described using primitive elements. However, in order to deal with real designs such as scan-based microprocessor designs, a test method should handle scan environment efficiently and certain logic blocks described in terms of their logical functions. Since a delay test seeks to determine whether a signal change propagates through a circuit within a given time period, it must provide at least two vectors. This makes delay testing in standard scan environment very difficult while the use of full-scan design can provide very substantial benefits for testing classical stuck-at faults.

There are several approaches to achieve high performance in path delay test generation. One approach is using enhanced-scan flip-flops where all the test generation methods devised for purely combinational circuits [1], [3]-[7]. However, this requires too much additional chip area. Therefore, test pattern generation for delay paths had to be performed in a standard scan environment. Since the initial vector of each two-vector path delay test could be scanned into the circuit's internal memory elements, however, all that was required to generate the tests was to be able to trace the test requirements through two clock cycles. This approach has been referred to as functional justification of the required machine state, and there have been several descriptions of how this might be accomplished [2], [8]-[13].

Additionally, in real designs, there are certain blocks of logic needed to be described by their logical functions rather than by circuit primitive elements. The reasons for this are two-fold. One is that the circuit timing analysis, which provided the list of paths to be tested, was to be performed with certain blocks not being expanded into primitive elements due to the blocks being custom-designed with non-standard logic devices. This means that the path descriptions would include only the block as a whole and no particular subpath through the block. Other blocks were implemented at the transistor level. This also resulted in certain blocks being present in the path descriptions as single entities rather than as collections of primitive sub-elements. For both types of blocks, the test generator needed to operate at the block level rather than at the level of circuit primitives. For this reason, the test generator had to perform on a multi-level circuit description, in which certain custom logic blocks were specified at the functional level. Handling these blocks for path delay testing is more difficult than for stuck-at testing. Using the ideas of stuck-at testing, functional level delay test pattern generation is proposed [14]. However, this approach is a non-enumerative automatic test pattern generation (APTG) and cannot guarantee the accurate testing of the actual path. There are several approaches for handling functional blocks using gate-equivalent descriptions [15]-[17]. In these approaches, hazard free tests are not guaranteed. In other words, the test patterns generated do not detect whether the path includes hazards or not. Also, since the requirements for path delay tests are over-constrained, a lot of test patterns may be generated to achieve the required fault coverage. Also, there are several approaches to handle IP blocks [18], [19]. However, these approaches use extra design for testability (DFT) circuitry to make delay testing possible.

The objective of this paper is to develop an efficient path delay fault test pattern generator for custom logic blocks. The test generator should be able to handle real circuits of as large as 100,000 logic devices. Our work was undertaken to provide the capability to perform robust path delay test generation under these rather stringent requirements. In the following sections, in order to handle real design with custom logic blocks efficiently, a new logic value system is introduced. And then delay testing issues for the custom logic blocks are explained followed by the implementation issues. Based on the test strategy and other observations, an efficient test pattern generation algorithm for custom logic blocks is developed. Finally, experimental results

and conclusions are provided.

II. LOGIC ALGEBRA

The design environment for developing an automatic test pattern generator for circuit delays is scan-based microprocessor design. In order to meet the requirements, the test generation time should be minimized and tri-state devices should be handled. Therefore, we develop a 29-valued logic system as depicted in Table 1. The first 16 values represent all the combinations of {0, 1, X, Z} in the two time frames, where Z means high impedance for the tri-state elements. The first character represents the value in the initial time frame and the second character represents the value in the final time frame. However, these values cannot represent presence or absence of hazards in the signals. Values {S0, S1, SZ} represent signals of {0, 1, Z}, respectively, in both time frames and also guarantee that they are free of hazards.

We add 10 additional values (from 00T to SB in Table 1) to the logic system. These values are especially useful in the test generation and used only for forward implication. They specify the unjustifiable values and hence help guide the justification process in the test generation. Instead of specifying what value a signal is, we can specify what values a signal cannot be. So, the justification process can avoid numerous unsuccessful searches earlier even at lines with unspecified values. Values {00T, 0XT, X0T, 11T, 1XT, X1T} are similar to values {00, 0X, X0, 11, 1X, X1}, respectively, but in addition the signals cannot be guaranteed to be free of hazards. Value XT0 (XT1) means unspecified value X in both time frames, but it cannot be S0 (S1). Value XTB means unspecified value X in both time frames, but it can be neither S0 nor S1. Complementarily, value SB means unspecified value X, but it can be either S0 or S1 only. Another new value SB is used in justification process for hazard custom-logic blocks only and so is not used in implication. See the discussion for custom-logic blocks for details. Finally, we have a 29-valued logic system, as depicted in Table 1.

There are three logic values that are logic-1 for both time frames: 11, S1 and 11T. The differences among them are related to the information that they can provide about static hazards in the signals. A logic value of 11 on a gate indicates that there are unspecified values in the input cone of the gate which makes it impossible to determine whether there may or may not be a static hazard in the signal. For example, if all inputs of an AND gate are XT1 (*i.e.,* they cannot be S1), it implies the output to be XT1. So, even though we have not determined the value at that output, we know S1 cannot be justified at that output. Consider the example shown in Fig. 1. Let's assume that the objective is to set *G* to S1. If we do not use extensive logic values (00T to SB), then *E* should be 1X. In that case, we may choose *E* instead of *F* and there is a conflict since *E* cannot have stable values because of already assigned values. However, in our logic value system, *E* is 1XT and when we make a choice between *E* and *F*, *F* is selected as the only alternative that can produce S0 and we can avoid an unnecessary conflict. As shown in this example, by using the 29-valued logic system, the test generation time can be reduced.

Table 1. 29-valued logic.

00	01	0Z	0X
10	11	1Z	1X
Z0	Z1	ZZ	ZX
X0	X1	XZ	XX
S0	S1	SZ	
00T	0XT	X0T	XT0
11T	1XT	X1T	XT1
XTB	SB		

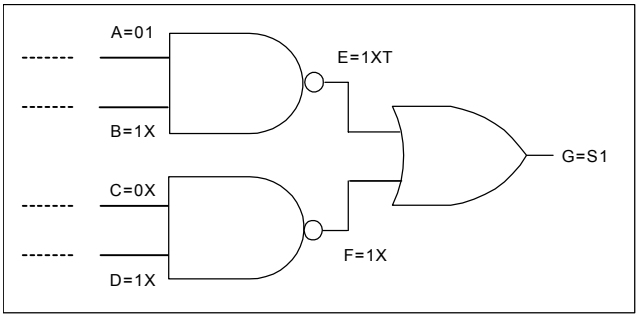


Fig. 1. Example of early conflict detection.

III. DELAY TEST STRATEGY FOR CUSTOM LOGIC BLOCKS

We define a *Custom Logic Block (CLB)* as a multi-input single-output device that realizes a particular combinational logic function. The output of a CLB is either logic-0 or logic-1; thus, it does not take a high-impedance (or tri-state) value. If there are complex logics that include sequential elements, these can be partitioned into sequential parts and combinational parts. If the complex block is provided without any information about sequential elements, it is almost impossible to test path delay faults since path delay tests require two patterns. Then, the combinational parts are further divided into small sub-blocks for each output. Also, in this way, a complex block which includes tri-state devices can be modeled. Therefore,

for complex logics, we can model them using CLBs. The complex CLBs and IP cores can be handled using the same modeling. In order to test delay faults of complex CLBs or IP-cores including sequential elements, their information should be provided. Otherwise, it is almost impossible to test for delay faults without DFT circuitry. All the approaches for delay testing of IP cores focus on design of extra logics that can help the testability of the IP cores. IP cores are divided into sequential parts and combinational parts. The complex combinational parts can be further divided into small blocks for each output. Using the new algorithm, test patterns for path delay faults are generated. An example is shown in the Fig. 2.

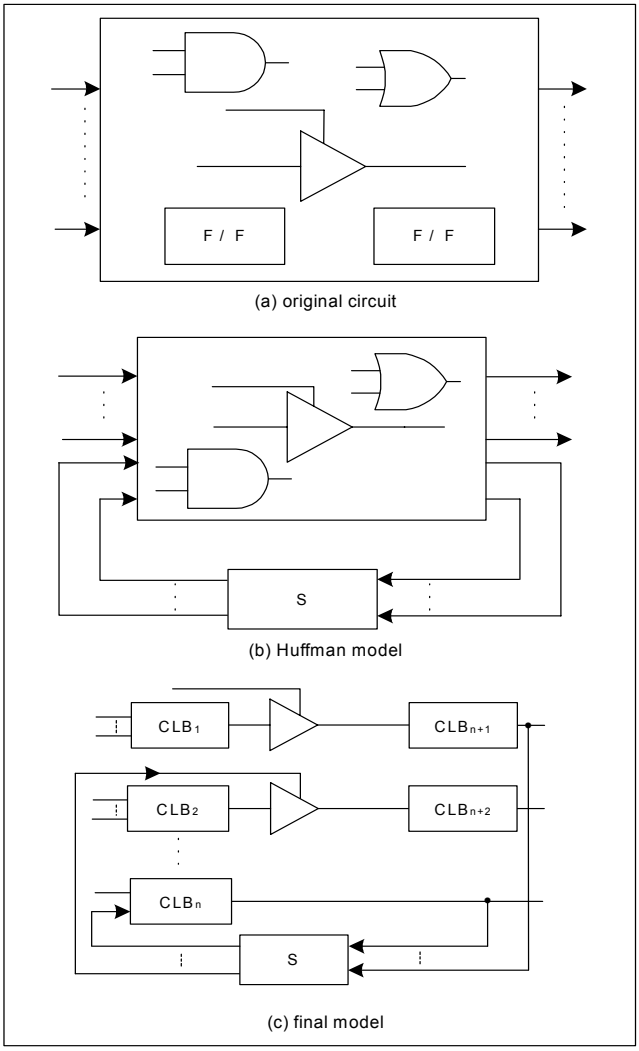


Fig. 2. Example of handling complex logic blocks.

The *initial value* of a line that is on the path being tested does not mean the actual value of the line determined by the first vector applied to the circuit but the value of the line during the first time frame for the specified transition (0 for a rising or 1 for a falling transition). The *final value* of a line on the path being tested is the opposite of its initial value. We call a CLB input with a specified transition (rising or falling) together with a specified transition for the output of the CLB a *CLB path.* We refer to the input of a CLB path as the *on-path CLB input.* All the other CLB inputs are referred to as *side inputs.* Let F denote the function realized by a CLB, and x_i denote an input of the CLB. A *non-inverting sensitization vector* for the x_i, $NI(F, x_i)$, is defined as a vector that satisfies the function $NI(F, x_i) = (\overline{F}|x_{i=0}) \cdot (F|x_{i=1})$. Similarly, an *inverting sensitization vector* for the x_i I, (F, x_i), is defined as a vector that satisfies the function $I(F, x_i) = (F|x_{i=0}) \cdot (\overline{F}|x_{i=1})$. In the following, when we say sensitization, we mean either non-inverting or inverting sensitization depending on the transitions specified for the CLB path.

In this paper, we define *a robust test for a CLB path* as a two-vector test which guarantees that the CLB output will not change from its initial value before the on-path input changes from its initial value independent of the delays inside the CLB or in the rest of the circuit, and which satisfies the specified initial and final values for the CLB path. A robust test for a CLB path might activate a set of internal paths starting from the on-path input, while another robust test for the same CLB path might activate a different set of internal paths again starting from the on-path input. Therefore, a robust test for a CLB path will detect a delay fault in a CLB if at least one internal path that can cause the CLB output make the specified transition has a delay fault, and for each unfaulted internal path that causes the CLB output make the specified transition, there is another unfaulted internal path that makes the output switch back to its initial value during the application of the test. A robust test for a CLB path might also detect a delay fault on an internal path starting from a side input. We define *a hazard-free robust test for a CLB path* as a robust test for the same CLB path such that the test does not cause a dynamic hazard at the output of the CLB. Therefore, a hazard-free robust test for a CLB path will detect a delay fault in a CLB if and only if all the internal paths along which the on-path input transition can travel to the output have delay faults on them. We define *a weak non-robust test for a CLB path* as a two-vector test such that the second vector is a sensitization vector for the on-path input of the CLB. We define *a strong non-robust test for a CLB path* as a weak non-robust test for the same CLB path such that the first vector makes the CLB output acquire its initial value.

Even though the necessary and sufficient side input constraints for simple gates are well-known for path delay-fault test generation, a mechanism is needed to automatically generate the necessary and sufficient side input constraints for CLBs which can perform arbitrary Boolean functions.

Let V denote a vector applied to the side inputs of a CLB.

We call V an *initialization vector* if and only if the CLB output acquires its initial value when V is applied, and the on-path CLB input has its initial value. We call an initialization vector V a *hazard-free initialization vector* if and only if a falling or a rising transition at a CLB side input which is X in V, does not cause a static hazard at the CLB output when all other side inputs are kept at some Boolean values that satisfy V and the on-path input has its initial value. We call a (hazard-free) initialization vector V a *minimal (hazard-free) initialization vector* if and only if there is no element x_j of V such that x_j is 0 or 1 in V, and when x_j becomes X, V is still a (hazard-free) initialization vector. Let V_1 denote a minimal hazard-free initialization vector for a CLB path, and let S denote the set of side inputs that are 0 or 1 in V_1. Let V_2 denote a sensitization vector such that any side input in S has the same value in V_2 as it has in V_1, and any side input that is 0 or 1 in V_2 but X in V_1 is necessary for V_2 to be a sensitization vector. Let $<V_1, V_2>$ denote a vector pair such that all side inputs in S are kept stable (glitch-free) during the transition from V_1 to V_2.

Theorem 1: A vector pair $<V_1, V_2>$ constructed as described above for a CLB path forms a necessary and sufficient condition for a robust test for that CLB path.

Proof: V_1 being a hazard-free initialization vector guarantees that the CLB output will not change from its initial value unless the on-path input changes. V_2 being a sensitization vector guarantees that the CLB output will acquire its final value when V_2 is applied, and the on-path input has its final value. This proves the sufficiency. If a side input that belongs to S is not kept stable, then that side input might take the opposite of the value it had in V_1 during the transition from V_1 to V_2 while the on-path input still has its initial value. As a result, either the CLB output switches to its final value or a transition on another side input causes a static hazard at the CLB output, because V_1 was minimal for hazard-free initialization. If a side input which is X in V_1 but 0 or 1 in V_2 becomes X, then V_2 is not a sensitization vector any more, and the CLB output might not stabilize to its final value when V_2 is applied and the on-path input has its final value. This proves the necessity. □

A sensitization vector V is a *hazard-free sensitization vector* if and only if the following two conditions are satisfied. First, a falling or a rising transition at a CLB side input which is X in V, does not cause a static hazard at the CLB output when all other side inputs are kept at some Boolean values satisfying V, and the on-path CLB input has its initial or final value. Secondly, the on-path CLB input transition from its initial value to its final value does not cause a dynamic hazard at the CLB output when all side inputs are kept at some Boolean values satisfying V. Also, a (hazard-free) sensitization vector V is a *minimal (hazard-free) sensitization vector* if and only if there is no element x_j of V such that x_j is 0 or 1 in V, and when x_j becomes X, V is still a (hazard-free) sensitization vector.

Let V denote a minimal hazard-free sensitization vector for a CLB path. Let $<V, V'>$ denote a vector pair such that all side inputs whose values are 0 or 1 in V have the same value in V' and remain stable (glitch-free) during the transition from the application of V to the application of V', and the on-path input has its initial value in V and its final value in V'.

Theorem 2: A vector pair $<V, V'>$ constructed as described above for a CLB path forms a necessary and sufficient condition for a hazard-free robust test for that CLB path.

Proof: Since a hazard-free sensitization vector is also a hazard-free initialization vector, $<V, V'>$ is a robust test for the CLB path. V being a hazard-free sensitization vector guarantees that no hazard is possible for the CLB output. This proves the sufficiency. Let's assume that a side input that is 0 or 1 in V is not kept stable. That side input might now take the opposite of the value it had in V during the transition from the application of V to the application of V', violating the hazard-free sensitization, because V was minimal. As a result, one of two things will happen. The one thing is that a side input transition will cause a static hazard or the on-path input transition will cause a dynamic hazard at the CLB output. The other is that the CLB output is not sensitized to the on-path input any more, and transitions at this side input together with the on-path input transition can cause hazards (glitches) at the CLB output. This proves the necessity. □

A minimal sensitization vector for a CLB path in the second time frame forms a necessary and sufficient condition for a *weak non-robust* test for that CLB path. A minimal sensitization vector for a CLB path in the second time frame together with a minimal initialization vector in the first time frame form a necessary and sufficient condition for a *strong non-robust* test for that CLB path.

IV. PATH DELAY TESTING FOR CUSTOM LOGIC BLOCKS

Analysis at the transistor level implementation of a CLB is needed to identify minimal hazard-free initialization and minimal hazard-free sensitization vectors. Due to difficulty of this task, we have decided to classify CLBs into two groups. If we have access to the transistor level schematic of a CLB, and it is not overly complex for manual analysis, such as a CMOS And-Or-Invert (AOI) gate or a CMOS 2×1 multiplexer, then we check whether all the minimal initialization vectors are also the minimal hazard-free initialization vectors, and all the minimal sensitization vectors are also the minimal hazard-free sensitization vectors for all possible CLB paths. If so, we call that CLB a

hazard-free CLB; otherwise, we call it a *hazard CLB*. If we do not have access to the transistor level schematic of a CLB, or it is overly complex for manual analysis, then we classify that CLB also as a *hazard CLB*. For a hazard-free CLB, we can use a minimal initialization vector for V_1 in the vector pair $<V_1, V_2>$ in Theorem 1, and we can use a minimal sensitization vector for V in the vector pair $<V, V'>$ in Theorem 2.

We assume the worst case for a hazard CLB in the sense that there might not be a hazard-free sensitization vector for any of its CLB paths. That is, even though all the side inputs are kept at stable values, a transition at the on-path input might cause a dynamic hazard at the CLB output. Therefore, we cannot guarantee a hazard-free robust test for a path that contains a hazard CLB, and we do not attempt to generate such a test. Again, if we assume the worst case for a hazard CLB, a hazard-free initialization vector cannot have any X value for a side input; otherwise, a falling or a rising transition at that side input might cause a static hazard at the CLB output when the on-path input still has its initial value. Thus, V_1 and V_2 in Theorem 1 must be vectors that are identical (except for the on-path input) and completely specified (no X value), and a minimal sensitization vector pair $<V, V'>$ modified to make every element of V whose value is X a stable 0 or a stable 1 becomes a necessary and sufficient side input constraint for a robust test for a CLB path of a hazard CLB. When a CLB is found to be untestable, what it means for the testability of its internal paths is given in Table 2.

Table 2. The relation between an untestable CLB and its internal paths.

	HFR	ROB	SNR	WNR
Hazard-free CLB	①	①	①	①
Hazard CLB	N/A	②	①	①

① If the CLB is untestable, then any internal path starting from the on-path input is untestable, as well.

② If the CLB is untestable, some internal paths starting from the on-path input may still be testable. If the CLB implementation is given, a robust or even an HF-robust test for such an internal path may be found.

We use an sum of products (SOP) representation such that every cube in F or in $\overline{F}$ is a prime implicant, all the prime implicants of F are included in the SOP representation for F, and all the prime implicants of $\overline{F}$ are included in the SOP representation for $\overline{F}$. We require that all prime implicants be included in the representations of the functions because we form the sensitizing functions for path transitions by combining a prime implicant of F with a prime implicant of $\overline{F}$, and we cannot guarantee that all possible sensitizing conditions are represented unless all prime implicants are represented for each of the functions. Therefore, the set of cubes in $F|x_{op=iv}$ or in $\overline{F}|x_{op=iv}$, where iv is the initial value of the on-path CLB input x_{op}, correspond to the set of all minimal initialization vectors when the initial value for the CLB output is 1 or 0, respectively.

We represent function $NI(F, x_{op}) = (\overline{F}|x_{op=0}) \cdot (F|x_{op=1})$ in SOP form constructed by performing pairwise ANDing of the cubes of $\overline{F}|x_{op=0}$ with the cubes of $F|x_{op=1}$, and eliminating every resulting cube that is covered by another cube generated in the same fashion. This way we insure that every cube in $NI(F, x_{op})$ is a prime implicant, and all the prime implicants of NI (F, x_{op}) are included in its SOP representation. Similarly, we construct $I(F, x_{op}) = (F|x_{op=0}) \cdot (\overline{F}|x_{op=1})$.

Theorem 3: The set of cubes in $NI(F, x_{op})$ or in $I(F, x_{op})$ correspond to the set of *all* minimal non-inverting sensitization vectors or *all* minimal inverting sensitization vectors, respectively, for CLB input x_{op}.

Proof: It is clear from the definitions of a prime implicant and a minimal sensitization vector, and the fact that $NI(F, x_{op})$ and $I(F, x_{op})$ consist of all their prime implicants. □

Theorem 3 is used for generating robust tests for hazard CLBs, hazard-free robust tests, and strong or weak non-robust tests. We actually may not generate all the prime implicants of $NI(F, x_{op})$ or $I(F, x_{op})$ at once, because we generate them on demand. That is, we generate a fixed number of additional prime implicants, if there is any left, only when x_{op} is the on-path input of the CLB, and all previously generated and stored prime implicants (minimal sensitization vectors) have been tried as side input constraints to realize that they are not justifiable. Since the overall test generation algorithm treats selecting a prime implicant (a minimal sensitization vector) as a decision the same way a primary input or a present state assignment is a decision, and there is a limit on the number of backtracks done in the decision stack, we do not need to set a separate limit on the number of prime implicants that will be generated. Since the new method gradually generates a prime to make test patterns, large circuits can be handled.

Let q_1 be a cube from $\overline{F}|x_{op=0}$ and q_2 be a cube from $F|x_{op=1}$. The *marked cube* $q_{1,2}$ is the union of q_1 and q_2, where every literal that is in q_2 but not in q_1 is marked. If we assume that $q_{1,2}$ is a prime marked cube, which will be defined shortly, $q_{1,2}$ represents a $<V_1, V_2>$ vector pair in Theorem 1 with V_1 being a minimal initialization vector. The unmarked literals in $q_{1,2}$ represent all the specified (0 or 1) elements of the minimal initialization vector V_1, which remain stable during the transition

tialization vector V_1, which remain stable during the transition from V_1 to V_2. The marked literals in $q_{1,2}$ represent all the additional specified side inputs, which are required only in the second time frame, that is, in the sensitization vector V_2.

A marked cube a_m *covers* a marked cube b_m if (*i*) the set of literals in a_m is a subset of the literals in b_m and (*ii*) every unmarked literal in a_m is also unmarked in b_m. A marked cube c_m is a *prime marked cube* if no other marked cube generated by some q_1, q_2 pair covers c_m. We represent function $RR(F, x_{op})$ in SOP form constructed by combining every possible q_1, q_2 cube pair as described above and eliminating every resulting marked cube $q_{1,2}$ that is covered by a marked cube generated earlier. This way we insure that every cube in $RR(F, x_{op})$ is a prime marked cube, and all the prime marked cubes of $RR(F, x_{op})$ are included in its SOP representation.

The *RR* symbol in $RR(F, x_{op})$ represents the case where the on-path CLB input and the CLB output are both rising. The other three combinations are as follows. Function $FF(F, x_{op})$ is generated by assigning cubes from $F|x_{op}=1$ to q_1 and assigning cubes from $\overline{F}|x_{op}=0$ to q_2. Function $RF(F, x_{op})$ is generated by assigning cubes from $F|x_{op}=0$ to q_1 and assigning cubes from $\overline{F}|x_{op}=1$ to q_2. Finally, function $FR(F, x_{op})$ is generated by assigning cubes from $\overline{F}|x_{op}=1$ to q_1 and assigning cubes from $F|x_{op}=0$ to q_2.

Theorem 4: The set of cubes in $RR(F, x_{op})$, $FF(F, x_{op})$, or in $RF(F, x_{op})$ or in $FR(F, x_{op})$ corresponds to the set of *all possible* <V_1, V_2> vector pairs in Theorem 1 with V_1 being a minimal initialization vector, for the case of input and output both rising, or input and output both falling, or input rising and output falling, or input falling and output rising, respectively.

Proof: It is clear from the definitions of a prime marked cube, a <V_1, V_2> vector pair, a minimal initialization vector, and the fact that $RR(F, x_{op})$, $FF(F, x_{op})$, $RF(F, x_{op})$, and $FR(F, x_{op})$ consist of all their prime marked cubes. □

Again, we actually may not generate all the prime marked cubes at once, but we generate them on demand similar to the generation of prime implicants for $NI(F, x_{op})$ or $I(F, x_{op})$. We used Theorem 4 for generating robust tests for hazard-free CLBs.

We would like to note that even though the SOP representation of a CLB may suggest a hazard condition, the actual implementation can be hazard-free. For example, consider a CLB with the function $\bar{s}\,a + s\,b$. If this is a hazard-free CLB, a test with $a = 1$, $b = 1$, $s = X$ as a first vector and $a = 0$, $b = 1$, $s = 0$ as a second vector, where b is kept stable, is a robust test for the CLB path with a falling and the output falling. But one may think that a falling transition at s when a is still 1 can create a 0-hazard at the output since there are two paths with opposite inversion parities from s to the output. This would be correct if the CLB was implemented using two AND gates, one for each product term, and an OR gate and an INVERTER. There exists, however, a different transistor-level implementation for this function to create a hazard-free CLB.

V. PATH DELAY TEST PATTERN GENERATION SYSTEM

In order to handle large circuits including custom logic blocks efficiently, an efficient test pattern generation system is developed which uses less memory spaces and reduces test generation time. The basic approach adopted for test generation is to consider each circuit partition as a combinational circuit spread over two time frames (clock cycles) since the logic values of the scan flip-flops in the first time frame are controllable, but in the second time frame, the values are determined by the function of the circuit. Sufficient time is allowed between the applications of the first vector and the second vector for all the circuit signals to stabilize to their steady-state values determined by the first vector. Given the size of circuits such as microprocessors, the approach should be the least memory intensive; the state transition is done by merely transferring logic values between the outputs and inputs of sequential devices.

The main path delay test generation algorithm is shown in Fig. 3. The testing process is performed to cover as much of the requirement space as possible. Therefore, our test strategy is as follows. We first attempt to generate a hazard-free robust (HFR) test for the specified delay path. If there exists a test, testing for this path is complete since none of the other tests could add to the requirements imposed by this test. Otherwise, we attempt to generate a robust (ROB) test. If it fails, the search for a strong non-robust (SNR) test is executed. If this is successful, the search for a test stops. Otherwise, we attempt to generate a weak non-robust (WNR) test.

Consider the "find_one_test" routine. As a first step, in "set_constraints" routine, according to the given test mode (hazard-free robust, robust, strong non-robust, or weak non-robust), required constraints are set to all side inputs along the path. Notice that the constraints are different for each test modes. As an example, constraints on side inputs of simple gates are shown in Table 3. All the constraints assigned are mandatory. Therefore, if there is a conflict in setting constraints on all side inputs, the path is untestable.

The next step is to perform backward implication. In the case of an output having stable value, 5 logic values {S0, S1, SZ, SB, XX} are considered. In the case of an output having other logic values, it can be treated as a 4 value logic {0, 1, X, Z} by

splitting the two-time-frame value into two one-time-frame values, implying twice, once for each time frame value, and merging the implied values over two time frames. For backward implication of custom logic blocks, if a custom logic block has a single literal product term in sum of product representation, it means that when the output is 0, backward implication is possible. It is obvious that if there is a single literal product term in sum of product representation, then all inverse product terms in inverse sum of product representation include this single literal. Therefore, by setting this literal, backward implication can be accomplished. If there is a single literal inverse product term, it means that when the output is 1, backward implication is possible. Notice that the output of an element cannot have the logic values representing stable impossible (such as 11T) as a justification objective. This backward implication is continued until no more implication is possible. During the procedure, if any implication is impossible because of a conflict with already assigned values, then it implies this path is an untestable path since these requirements are all mandatory. Also, during the backward implication procedure, when a primary input or an output of a scan-flop is set to non-X value, simulation (forward implication) is performed using the implied value.

After backward implication, justification of remaining objectives is performed by a state space search over circuit inputs and/or state variables. Since the choice is not necessarily mandatory, if there is a conflict, all available choices are considered to resolve the conflict. If no choice is available, the path is untestable. In order to reduce the search space, stable values are justified first. Then second time frame values are justified. Finally, first time frame values are justified. During the justification, when it encounters a flip-flop at the second time frame, the flip-flop output value of the second time frame is copied to the flip-flop input value of the first time frame. When it meets the flip-flop at the first time frame, the justification stops. Some examples are shown in Fig. 4.

```
test_paths()
{
   for(i=0;i<Num_paths;i++) {
      path = read_paths(i) ;
         /* find a hazard-free robust test */
         flag = find_one_test(path,HFR);
         if(flag!=TEST_GENERATED) {
            /* find a robust test */
            flag = find_one_test(path,ROB);
            if(flag!=TEST_GENERATED) {
               /* find a strong non-robust test */
               flag = find_one_test(path,STR);
               if(flag!=TEST_GENERATED) {
                  /* find a weak non-robust test */
                  flag = find_one_test(path,WEA);
               }
            }
         }
      }
}
find_one_test(path,test_mode)
{
   /* put constraints according to test types */
   flag = set_constraints(path,test_mode);
   if (flag==UNTESTABLE||flag==ABORTED)
      return(flag);
   /* backward implication */
   flag = backward_implication();
   if (flag==UNTESTABLE||flag==ABORTED)
      return(flag);
   /* justify the values */
   flag = make_choice();
   if (flag==UNTESTABLE||flag==ABORTED)
      return(flag);
   else
      /* test is generated */
      mark_success(path);
}
```

Fig. 3. Path delay test generation algorithm.

Table 3. Required constraints on side inputs.

Elements	On-path Transition	Side Input Contraints			
		HFR	ROB	SNR	WNR
AND/ NAND	Rising	S1	X1	X1	X1
	Falling	S1	S1	11	X1
OR/ NOR	Rising	S0	S0	00	X0
	Falling	S0	X0	X0	X0

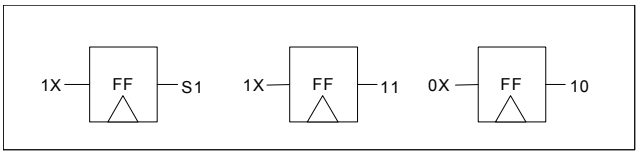


Fig. 4. Examples of state transition.

In order to achieve the maximum efficiency, stable values are justified first followed by the second and first time frame values. There is a list of elements whose required value is a stable value. In justification, if there are custom logic blocks, they are justified first. Since there are usually more than one combinations of fanin values which can set the required value at the output of the custom logic block, it is advantageous to justify custom logic blocks first. This justification is continued until the list is empty, the path is proved to be untestable, or the search is aborted. If the output value of the element is the same as the needed value, then the element is removed from the list and the next element is considered. If the value is different, in order to produce the needed value, trace is back to primary input or memory element. After assigning the proper value to primary input or memory element, forward implication is per-

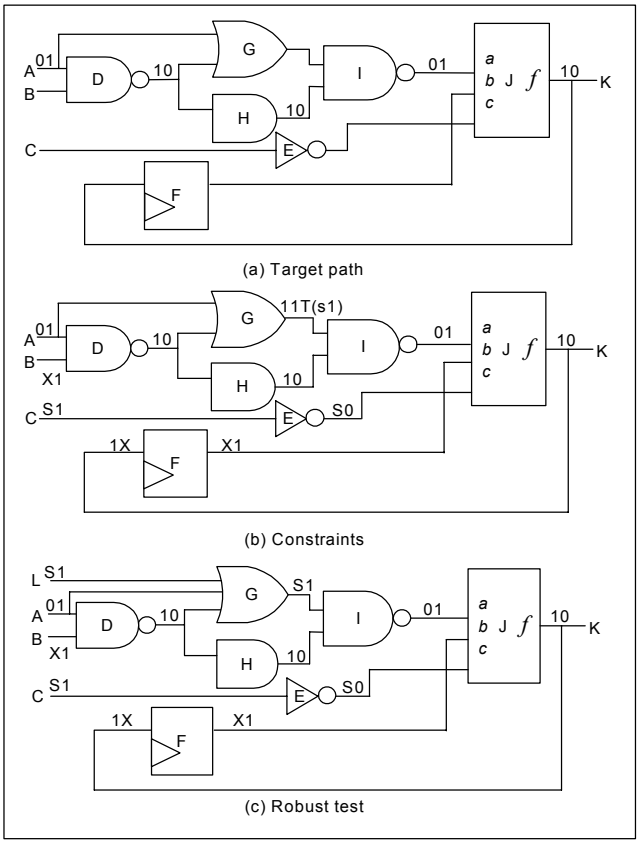


Fig. 5. Example of setting constraints for robust tests.

formed. If there is no conflict during the forward implication, then the next element is considered. However, if there is a conflict, another assignment is applied to resolve the conflict in the backup routine. The backup routine is similar to the backtrack in PODEM. If there is no remaining optional assignment, the path is untestable. In justification of the second time frame values, in order to reduce the search space, justification is done using 4 logic values {0, 1, X, Z} which are extracted from two-time-frame logic values. After justification, the new assignment is merged with the first time frame value. The procedure is almost the same as the stable value justification procedure except that the assigned value is second time frame value and when there is no other remaining assignment to resolve a conflict, it is returned to stable value justification procedure. In justification of the first time frame values, the procedure is almost the same as the second time frame value justification procedure except that the assigned values are first time frame values and when there is no other remaining optional assignment to resolve a conflict, we return to the second time frame value justification procedure. When the list is empty, all the requirements for the test have been satisfied and the path is tested.

To illustrate the algorithm, a simple example in Fig. 5 (a) is

Table 4. Circuit characteristics.

Circuit	Flip-Flops	Gates	Custom Blocks	Paths
s1494	6	647	0	1,000
s5378	179	2,779	0	1,000
s9234	228	5,597	0	1,000
s13207	669	7,951	0	1,000
s15850	597	9,772	0	1,000
s35932	1,728	16,065	0	1,000
s38417	1,636	22,179	0	1,000
s38584	1,452	19,253	0	1,000
Circuit 1	880	4,391	2,179	106
Circuit 2	2,152	42,216	10,655	332
Circuit 3	362	1,824	835	109
Circuit 4	3,922	32,455	8,935	1,000
Circuit 5	1,320	6,554	3,363	31
Circuit 6	3,160	36,318	9,835	379

used. There is a scan flop (*F*) and a custom logic block (*J*) whose function is given by $f = \bar{a}\,\bar{c} + \bar{b}\,\bar{c}$. In this example, let the path be {*A*(*rising*), *D*(*falling*), *H*(*falling*), *I*(*rising*), *J*(*falling*), *K*(*falling*) }, and the test type be robust test. Since the on-path fanin is *a* of the custom logic block (*J*), the requirement for robust test is *b* = X1 and *c* = S0. The first step is to set the constraints on all side inputs of the path. Therefore, for robust test, the following conditions should be satisfied: *B* = X1, *C* = S1, *G* = S1, *E* = S0 and *F* = X1. After backward implication of constraints, forward implication is performed for the assigned values of primary inputs or memory elements. As a result, *G* is 11T and there is a conflict since the objective of *G* is S1 as shown in Fig. 5 (b). Since we cannot make G to be S1, the path is untestable for robust test. However, we can generate a strong non-robust test for the path. The strong non-robust test is $<V_1, V_2>$ = <0X1X, 111> where the first vector contains the values of primary inputs (*A, B, C*) and scan flip-flop (*F*), and the second vector contains the values of primary inputs only. Consider the same path for robust test using a circuit in Fig. 5 (c) which is a slight modification of Fig. 5 (a). The difference is that the element *G* has another input *L*. After forward implication, *G* = 11. The next step is to justify the steady values. In order to make *G* = S1, *L* should be set to S1. Then, justification of second time frame value is performed followed by justification of first time frame value. Since there is no conflict, a robust test is found, that is, $<V_1, V_2>$ = <0X11X, 1111> where the first vector is the values of primary inputs (*A, B, C*, *L*) and scan flip-flop (*F*) and the second vector is the values of primary inputs.

Table 5. Test generation results.

Circuit	Robust			Strong Non-robust			Weak Non-robust			CPU (sec)	Memory (Mbytes)
	DP	UP	AP	DP	UP	AP	DP	UP	AP		
s1494	345	655	0	94	561	0	371	190	0	397.39	0.96
s5378	485	234	281	9	216	290	34	182	290	18482.31	1.62
s9234	0	1000	0	0	1000	0	0	1000	0	16.39	1.93
s13207	0	1000	0	0	1000	0	0	1000	0	21.35	2.55
s15850	0	1000	0	0	1000	0	0	1000	0	32.44	3.34
s35932	0	1000	0	31	969	0	341	628	0	523.96	4.05
s38417	185	732	83	10	804	1	345	459	1	1866.66	5.62
s38584	58	754	188	34	851	57	16	836	56	15864.43	4.83
Circuit 1	52	36	18	0	32	22	4	21	29	1978.76	7.81
Circuit 2	81	100	151	0	97	154	6	97	148	4906.87	22.52
Circuit 3	35	73	1	0	73	1	4	69	1	8.31	6.21
Circuit 4	79	253	668	9	246	666	123	100	689	11856.57	27.41
Circuit 5	0	31	0	0	31	0	0	31	0	880.86	11.28
Circuit 6	229	96	254	1	91	258	0	1	348	4025.26	21.27

DP: Detected Path, UP: Untestable Path, AP : Aborted Path

The requirement hierarchy, along with the consideration of the detection capabilities of each type of test, enabled us to devise a particular strategy for path delay test generation. The general strategy is to maximize the coverage of the requirement space with the exception that there would be no attempt to generate hazard-free robust tests for the purpose of screening parts. In cases where tests were being generated for part screening, the default option would be to generate robust tests. If this effort failed for a particular path, an attempt would be made to generate strong non-robust test for that path. In the event it proved to be impossible to generate a strong non-robust test for a path, then we would attempt to generate a weak non-robust test. If the purpose of the test set was delay defect diagnosis rather than part screening, then we would attempt to generate hazard-free robust tests as a first option.

VI. RESULTS

The algorithm described in the previous sections has been implemented in approximately 30,000 lines of C language. The set of benchmark circuits for the program included a sample of circuits from the standard ISCAS 89 benchmark set [20], as well as a set of circuits from a microprocessor.

Some characteristics of the circuits tested are given in Table 4. In this table, the number of flip-flops, the number of custom logic blocks, the number of gates, and the number of paths are given. For these circuits, a set of the longest paths were selected using a commercial timing analyzer since considering all possible paths for large circuits is almost impossible.

The program was tested with the benchmark circuits on a SUN Sparc10, with 32 Mbytes of main memory and a swap space of 157 Mbytes.

The path delay test generation results are shown in Table 5. In this table, the numbers of detected paths, untestable paths, and aborted paths for each type of test, CPU time, and memory usage are given. Since tests were generated for part screening, hazard-free robust tests are not considered in this result. Notice that if a path is detected by higher-constraint test, the path is no longer considered. Notice that for s9234, s13207, and s15850 circuits, there exists no test for any of the 1000 longest paths. In other words, all 1000 longest pats are false paths. For these results, the number of backtracks per path of each type were set to 10000.

The figures for CPU time and memory usage for our program and for Dsteed [8], [9] are given in Table 6. The reason why we compare our path delay test generator with Dsteed, is that it is only known path delay test generator for standard scan designs and other works use different environments. Since Dsteed generates only one type of test on each program invocation and it cannot generate robust tests, we have included a column giving the CPU time for our program when it was requested to generate only strong non-robust test (approximately

robust test) in order to provide a more direct comparison. As can be seen from the table, the CPU times are smaller than for Dsteed for large circuits. Recall that the primary motivation for the choice of a justification procedure was the minimization of memory usage. There is also a dramatic difference in memory usage, which appears to increase with increasing circuit size. As the number of flip-flops increases, the memory usage of Dsteed increases dramatically but our test generator is less sensitive. This difference, combined with our inability to run Dsteed on the three largest circuits in the benchmark set, provides some additional justification for our decision to adopt a less memory-intensive approach.

Table 6. Comparison with Dsteed.

Circuit	Our work		Dsteed	
	CPU Time	Memory Usage	CPU Time	Memory Usage
s1494	121.82	0.97	129.0	0.70
s5378	13522.86	1.58	23124.1	12.40
s9234	7.52	1.90	6.4	17.37
s13207	10.87	2.64	1316.1	24.23
s15850	13.97	3.22	10255.9	24.00
s35932	199.04	4.15	*	*
s38417	137.86	5.71	*	*
s38584	4425.48	4.77	*	*

* Memory exhausted after using just over 200 Megabytes

VII. CONCLUSION

In this paper, we have presented a path delay test generator for circuits having custom logic blocks with functional descriptions only. The approach is based on the combination of the limited time-frame expansion approach and the test generation approach for combinational circuits. Experimental results show significant improvement in memory and CPU usage of the proposed algorithm, compared with previous researches. In addition, the new test algorithm can be applied to designs employing IP circuits whose implementation details are either unknown or unavailable.

REFERENCES

[1] A. Pramanick and S. Reddy, "On the Detection of Delay Faults," *Proc. of Int'l Test Conf.*, 1988, pp. 845-856.

[2] K. Cheng, S. Devadas, and K. Keutzer, "Delay Fault Test Generation and Synthesis for Testability under A Standard Scan Design Methodology," *IEEE Trans. on CAD*, Aug. 1993, pp. 1217-1231.

[3] M. Schulz, F. Fink, and K. Fuchs, "Parallel Pattern Fault Simulation of Path Delay Faults," *Proc. of Design Automation Conf.*, 1989, pp. 357-363.

[4] G. Smith, "Model for Delay Faults Based Upon Paths," *Proc. of Int'l Test Conf.*, 1985, pp. 342-349.

[5] S. Reddy, C. Lin, and S. Patil, "An Automatic Test Pattern Generator for the Detection of Path Delay Faults," *Proc. of Int'l Conf. on Computer Aided Design*, 1987, pp. 284-287.

[6] C. Lin, "On Delay Fault Testing in Logic Circuits," *IEEE Trans. on CAD,* Sept. 1987, pp. 694-703.

[7] A. Pramanick and S. Reddy, "On Multiple Path Propagating Tests for Path Delay Faults," *Proc. of Int'l Test Conf.*, 1991, pp. 393-402.

[8] K. Cheng, S. Devadas, and K. Keutzer, "Partial Enhanced Scan Approach to Robust Delay Fault Test Generation for Sequential Circuits," *Proc. of Int'l Test Conf.*, 1991, pp. 403-410.

[9] K. Cheng, S. Devadas, and K. Keutzer, "Robust Delay Fault Test Generation and Synthesis for Testability under a Standard Scan Design Methodology," *Proc. of Design Automation Conf.*, 1991, pp. 80-86.

[10] S. Kang, B. Underwood, and W. Law, "Path Delay Fault Simulation for a Standard Scan Design Methodology," *Proc. of Int'l Conf. on Computer Design*, 1994, pp. 359-363.

[11] B. Underwood, S. Kang, W. Law, and H. Konuk, "Fastpath : A Path Delay Test Generator for Standard Scan Designs," *Proc. of Int'l Test Conf.*, 1994, pp. 154-163.

[12] T. Chakraborty, V. Agrawal, and M. Bushnell, "Delay Fault Models and Test Generation for Random Logic Sequential Circuits," *Proc. of Design Automation Conf.*, 1992, pp. 165-172.

[13] K. Fuchs, H. Wittmann, and K. Antreich, "Fast Test Pattern Generation for All Path Delay Faults Considering Various Test Classes," *Proc. of European Test Conf.*, 1993, pp. 89-98.

[14] M. Michael and S. Tragoudas, "Functional Based ATPG for Path Delay Faults," *Southeast Symp. on Mixed Signal Design*, 2000, pp. 159-164.

[15] I. Pomeranz and S. Reddy, "Functional Test Generation for Delay Faults in Combinational Circuits," *ACM Trans. on Design Automation of Electronic Systems*, Apr. 1998, pp. 231-248.

[16] I. Pomeranz and S. Reddy, "Functional Test Generation for Delay Faults in Combinational Circuits," *Proc. of Int'l Conf. on Computer Aided Design*, 1995, pp. 687-694.

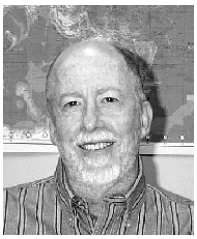
[17] W. Ke and P. Menon, "Synthesis of Delay Verifiable Combinational Circuits," *IEEE Trans. on Computers*, Feb. 1995, pp. 213-222.

[18] H. Kim and J. Hayes, "Delay Fault Testing of IP-Based Designs Via Symbolic Path Modeling," *Proc. of Int'l Test Conf.*, 1999, pp. 1045-1053.

[19] D. Nikolos, T. Haniokatis, H. Vergos, and Y. Tsiatouhas, "Path Delay Fault Testing of ICs with Embedded Intellectual Blocks," *Proc. of VLSI Test Symp.*, 1999, pp. 112-116.

[20] F. Brglez, D. Bryan, and K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," *Proc. of Int'l Symp. on Circuits and Systems*, 1989, pp. 1929-1934.

Sungho Kang received the B.S. degree from Seoul National University, Korea and received the M.S. and Ph.D. degrees in electrical and computer engineering from the University of Texas at Austin. He was a post doctoral fellow at the University of Texas at Austin, a research scientist at Schlumberger Laboratory for Computer Science, Schlumberger Inc., and a senior staff engineer at the Semiconductor Systems Design Technology, Motorola Inc. Since 1994, he has been an associate professor of Department of Electrical and Electronic Engineering at Yonsei University, Korea. His current research interests include VLSI design, VLSI CAD, VLSI testing and design for testability.



Bill Underwood received B.A. and M.A. degrees in mathematics, as well as an M.S.E. degree in electrical and computer engineering, from the University of Texas at Austin. In addition, he has received a Ph.D. in psychology from Stanford University. Dr. Underwood has worked at Motorola, the Microelectronics and Computer Technology Corporation (MCC), and Viewlogic Systems. He has held faculty positions at the University of Texas at Austin and Boston College. He holds several patents in the field of EDA for test and he has received a Best Paper award from the Design Automation Conference. He is currently employed as a Senior R&D Staff Engineer at Synopsys, where he works in automatic test pattern generation, fault simulation, and delay fault testing.

Wai-On Law received the B.S. degree in computer science and electrical engineering and the M.S. degree in electrical engineering from the University of Wisconsin at Madison in 1986 and 1987, respectively. He has worked in Motorola Semiconductors since 1988. Currently, he is with the Transportation and Standard Products Group, working on 32-bit microcontroller designs. His interests include VLSI test, test generation, design for testability, and microcontroller design.

Haluk Konuk received the B.S. degree from Bogazici University, Istanbul, Turkey in 1986, the M.S. degree from Case Western Reserve University in 1989, and the Ph.D. degree from the University at Santa Cruz in 1996. From 1989 to 1991, he was with CAD Language Systems Inc., Rockville, MD. He currently works in the California Design Center of Agilent Technologies, which was spun off in 1999 from Hewlett-Packard Co., Palo Alto, CA.