JOURNAL OF ELECTRONIC TESTING: Theory and Applications 10, 231–244 (1997)
© 1997 Kluwer Academic Publishers. Manufactured in The Netherlands.

Hierarchical Delay Test Generation

C.P. RAVIKUMAR
Department of Electrical Engineering, Indian Institute of Technology, New Delhi 110016, INDIA
rkumar@ee.iitd.ernet.in

NITIN AGRAWAL*
S3 India, 5th Floor, Prestige Meredian, M.G. Road, Bangalore 560001, INDIA
bewad@cyberspace.org

PARUL AGARWAL*
Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan, 48109, USA
pagarwal@eecs.umich.edu

Received July 19, 1995; Revised September 6, 1996 and October 28, 1996

Editor: K. Kinoshita

Abstract. Delay testing is used to detect timing errors in a digital circuit. In this paper, we report a tool called **MODET** for automatic test generation for path delay faults in *modular* combinational circuits. Our technique uses precomputed robust delay tests for individual modules to compute robust delay tests for the module-level circuit. We present a *longest path theorem* at the module level of abstraction which specifies the requirements for path selection during delay testing. Based on this theorem, we propose a path selection procedure in module-level circuits and report efficient algorithms for delay test generation. **MODET** has been tested against a number of hierarchical circuits with impressive speedups in relation to gate-level test generation.

Keywords: delay test generation, hierarchical testing, path selection

1. Introduction

For today's high speed integrated circuits, it is as important to verify the timing behaviour of a circuit as it is to test the circuit for static faults such as the popular "stuck-at" faults. The gate-delay and the path-delay fault models were introduced [1, 2] to test for the presence of timing faults in the circuit due to which the circuit may fail to function properly at the design clock speed. Between these two fault models, the path delay fault model has prevailed as the more popular model and is used in this paper; a path delay fault is characterized by a tuple $\langle p, \delta \rangle$, where p is a path from a primary input of the circuit to a primary output, and δ is the *type* of transition that must be propagated along path p. The transition type can be a falling transition $\downarrow$ or a rising transition $\uparrow$. A path delay is said to have occurred on path p if the transition of type δ fails to propagate across the path within time equal to the clock period. A pair of test vectors must be applied to test for a path delay fault [3]. Many efficient algorithms have been reported in the literature for generation of delay tests [3–5]. However, generating delay tests for a circuit of VLSI complexity can be quite time consuming due to several reasons (a) The number of paths, and

*Formerly with the Indian Institute of Technology, New Delhi, where this work was carried out.

hence the number of path delay faults, can be excessively large (b) The test generation for a single delay fault can be time consuming since most test generators used PODEM-like [6] backtracking algorithms and (c) When the circuit description is available at block-level (or macro-level), an overhead is encountered in the flattening of the netlist to gate-level. The situation can be improved in the following ways. Efficient *path selection* algorithms [7, 8] can be used to reduce the size of the fault set. Improved delay test generators (such as the FAN-based algorithm described in [4] or the algorithm based on binary decision diagrams described in [9]) can reduce the test generation time by reducing the number of backtracks or through the use of algebraic techniques). To alleviate the problem of flattening the hierarchical netlist, we propose in this paper a scheme to perform delay test generation at the module level itself. A *module* in this paper is a combinational logic block consisting of one or more logic gates; examples of modules are full adders, multiplexers, comparators, and so on. Our experimental results show that hierarchical delay testing can provide orders of magnitude speedup over gate-level test generation. To provide further speedups, we also report a path selection algorithm which works on hierarchical netlists.

Hierarchical test generation has been known for stuck-at faults. Well known test generation algorithms such as the D-algorithm and PODEM [10] have been extended to work for module-level circuits. As part of the OASIS design automation package, a test generator for stuck-at faults, called MODEM, is available [11]. MODEM, or module-oriented decision making, is an extension of PODEM and is applicable for circuits which contain small macros such as EXOR gates. Calhoun and Brglez observed an interesting fact with respect to hierarchical testing [11]. The authors classified stuck-at faults into module-internal and module-external faults, depending on whether a fault appears on a line which is internal to a module or on a line which connects two or more modules. It was observed that tests which were generated with module-external faults as targets also tend to cover most of the module-internal faults as well. Patel and Chandra [12] presented a hierarchical test generator called HIPODEM which also uses a branch-and-bound algorithm similar to PODEM [6]. In tune with the observation of [11], HIPODEM considers only module-external faults during test generation. Murray and Hayes proposed a technique called *PathPlan*, in which test data for modules is described using precomputed stimulus/response pairs. The stimulus/response files are processed symbolically using techniques derived from artificial intelligence. The test generation algorithm using in PathPlan is a variant of the D-algorithm.

Sarfert et al. [13] present a hierarchical test generation scheme for stuck-at faults in large combinational circuits. The authors distinguish between high-level primitives (HLP) and gate-level representations (GLR) of the primitives. Thus a k-input multiplexer can be viewed as an HLP or a black box with k data inputs, $\lceil \log_2 k \rceil$ control inputs, and a single output. An HLP may be associated with several GLRs. For instance, a GLR associated with 2-input multiplexer consists of two two-input AND gates and one two-input OR gate. Stuck-at faults that are internal to a GLR can be *transferred* to the inputs of the corresponding HLP by considering fault collapsing rules such as fault equivalence and fault dominance. However, some faults internal to a GLR cannot be transferred to the inputs of the corresponding HLP; in a hierarchical representation of the circuit, such faults will remain *internal* to the corresponding HLP. Tests can be generated for faults which are at the boundaries of an HLP without flattening the HLP to its GLR. On the other hand, the ATG algorithm of [13] expands an HLP X to its GLR if there are internal faults associated with *X only when the internal faults of X are under consideration*. Thus the speed and memory advantage in [13] comes predominantly due to this *selective flattening*. The strategy in **MODET** differs from [13] in a fundamental way in that, during test generation, flattening of a high-level primitive to its gate-level representation is unnecessary. In **MODET**, we find a sufficient number of path delay faults at the *hierarchical level* based on a *longest path theorem* as will be explained in Section 2.

Recently, Pomeranz and Reddy [14] have reported a test generator for circuits composed of macro blocks when the internal gate-level implementation of the macros are unknown. Thus their technique is useful in technology-mapped circuits implemented as FPGAs or PLAs; it is assumed that the truth table of the macro is known and the number of outputs per macro is restricted to 1. In our work, we assume that gate-level realization of modules is known through a library, and do not place any restrictions on the number of inputs and outputs of a macro. Pomeranz and Reddy do not use the conventional path delay fault model for macro-level circuits since the internal details of macros are unknown; they introduced new fault models analogous to gate delay and gate-level path delay fault models.

In comparion, our hierarchical delay tester as well as the hierarchical path selection algorithms deal with the conventional gate-level path delay fault model, and it is therefore easier to interface our tools with existing delay test software such as delay fault simulators.

This paper is organized as follows. In the next section, we describe the notion of *module-level paths* in a modular circuit and discuss a hierarchical path selection algorithm. In Section 3, we describe **MODET**, our module-level test generator for combinational circuits. We have implemented **MODET** on Sun/SPARC workstations; Section 4 describes experimental results on a number of common hierarchical circuits. Section 5 provides conclusions.

2. Module-Level Path Selection

2.1. Assumptions

In our present implementation, each gate is assumed to have one unit delay, and the rise delay of each gate output is assumed to be equal to its fall delay. However, the above two assumptions can be easily relaxed by appropriately modifying the cell library information. We assume that apart from a hierarchical netlist of the circuit, we are also given a netlist description of each module type. Thus our delay test generator assumes the presence of a *module library*; for each module in the library, we also assume that the following information is available.

1. Every module-internal path is described in terms of module-internal names.
2. Test vectors are provided for every module-internal path.
3. Total delay for each module-internal path is available.

The above assumptions are realistic, since gate-level descriptions for standard MSI-level library components (such as full adders, multiplexers, decoders, comparators, and so on) are available from the vendors. Since the modules are small in size, it is not expensive to run a gate-level delay test vector generator to precompute the test vectors for the module-internal paths. In fact, our hierarchical delay tester can itself be used to synthesize the module information by selecting logic gates as the building blocks. The storage overhead associated with the cell library is not significant in comparison to the amount of main memory that would have otherwise been required had we considered a flattened netlist. The computational effort in synthesizing the library is also not significant, since it is a one-time effort.

We employ 5-valued logic (see [3]) in our test generation algorithm. Such a logic system consists of five logic values, namely, $S0$ (static 0), $S1$ (static 1), $U0$ (transition from unknown value U to 0 if the corresponding line is not on the path being tested, or transition from 1 to 0 if the line is on the path being tested), $U1$, and XX (don't care). We generate robust tests for each of the selected path delay faults in the module-level circuit, provided such a test exists for the fault under consideration.

2.2. Modular Fault Model

Given the gate-level realization of a module M, we refer to a path in module M as a module-internal path. Given a modular circuit, a module-level path is viewed as a concatenation of several module-internal paths. Consider the example of Fig. 1 which shows a hierarchical circuit composed of two modules $M1$ and $M2$. The path P highlighted in the figure is

$$P = (X_1, x_1, x_5, x_7, X_5, y_1, y_4, y_6, X_7).$$

P can be viewed as a chain of two module-internal paths, namely, $P_1 = (X_1, x_1, x_5, x_7, X_5)$ through $M1$ and $(X_5, y_1, y_4, y_6, X_7)$ through $M2$. We now introduce the notion of *virtual fanouts* in the module-level representation of circuit; if there are β module-internal paths from an input pin a to an output pin b of a module, we abstract these paths in the form of "virtual fanouts" at pin a. In the above example, there are 3 paths from pin X_5 to pin X_7 in module $M2$. Thus, in the abstract representation of the modular circuit (see Fig. 2) we

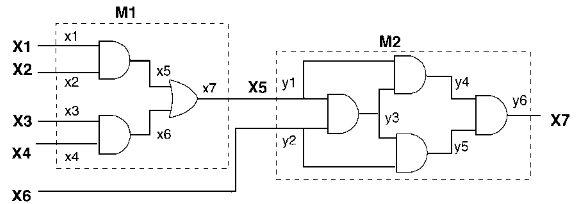

Fig. 1. A path in a flattened module-level circuit.

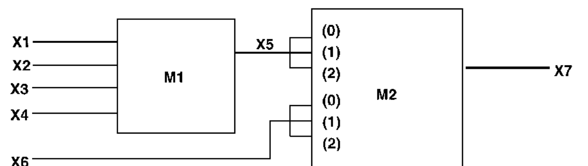

Fig. 2. Modular circuit with virtual fanouts.

show three fanouts at the pin X_5. The module-level path P in the abstract representation will be simply denoted as $P = X_1(0), X_5(1), X_7(0)$, where the notation $y(i)$ denotes the ith virtual fanout of net y. It is easy to see that a module-level path Q can be uniquely constructed for a gate-level path P in the flattened version of the modular circuit. The user of our modular-delay tester must specify a path delay fault in the form of a tuple $\langle$ *path*, *fault type* $\rangle$, where *path* is a module-level path and *fault type* $\in \{\uparrow, \downarrow\}$.

2.3. Path Selection

In a multiple-input, multiple-output combinational circuit, the total number of paths can be exponential in the number of signal lines in the circuit (see Section 4). It is impractical to consider all path delay faults during delay testing. Li et al. gave algorithms for *path selection* in gate-level combinational circuits [8]. As the cost of generating robust delay tests for a circuit depends on the number of paths in the fault set, it is desirable that the selected set of paths SP be as small as possible. At the same time, it is important that the selected paths are fairly representative of all the paths in the circuit, and, if a delay fault exists in the circuit then we are more likely to detect the fault by testing for only the selected paths. The most popular way to perform path selection [8] at gate level is to select a set of paths SP in the logic circuit such that for each lead l in the given circuit C there is at least one target path in SP, whose propagation delay is no less than the delay of any other path containing the lead l. Besides reducing the number of paths to manageable proportions, this strategy also generates a wide spectrum of paths from the point of view of testability and the propagation delays. Based on this criteria for path selection a *longest path theorem* has been derived in [7] which states.

Theorem 1 (Heragu et al.). *In a circuit, the minimum set of paths from inputs to outputs such that each circuit lead is included in at least one path whose propagation delay is no less than the delay of any path containing the lead comprises the longest path through* (a) *all primary inputs and* (b) *all fanout branches except the ones having the highest level number* (*with respect to a primary output*) *at their fanout stem.*

We have implemented two approaches for performing path selection at a *hierarchical level.* The first technique involves flattening of the module-level description to gate-level, performing path selection at the gate-level in accordance to Theorem 1, and finally interpreting the gate-level paths as module-level paths. The second approach (called Approach 2 hereafter) works directly on a hierarchical netlist, using precomputed information about module-internal paths. Our path selection algorithms can handle mixed-level circuits containing modules as well as logic gates.

The flattening procedure required in our first approach (denoted by Approach 1 hereafter) generates a gate-level netlist from the given hierarchical netlist, while retaining the line numbers of the module-level nets in the corresponding gate-level circuit. Such a numbering is useful when the selected paths are to be mapped back to the module-level. The gate-level path selection procedure has two preprocessing steps, namely *Mark_Depth* and *Mark_Level*. The former procedure traverses the circuit graph from primary outputs to the primary inputs, marking depths for each line. The depth of a line l is defined as the maximum distance of l from any primary output. The latter procedure makes a forward pass through the circuit graph and marks levels for each line l; the level of a line l is the maximum distance from any primary input to line l. These procedures are described in detail in [15]. After the preprocessing, we select longest delay paths through every primary input and every fanout branch except the ones having the highest depth at the fanout stem. The gate-level paths are then mapped to module-level paths using the line number information stored during the flattening step. In our experimentation, we found that the Approach 1 for path selection is inefficient (see Section 4); we therefore developed a path selection algorithm which works directly on module-level netlists.

2.4. Module-Level Path Selection

Path selection at module-level makes use of the concept of virtual-fanouts introduced earlier (Section 2.2). Three preprocessing procedures are involved, namely, *Virtual_Fanouts*, *Mark_Level* and *Mark_Depth*. The first procedure creates virtual fanouts for every input line l of every module M if the number of module-internal paths from l to any of the output lines of M is more than one. The procedures for marking the level and the depth for the lines in the module-level circuit are shown in Figs. 3 and 4.

The procedure *Mark_Depth_Hierarchical* shown in Fig. 3 uses a breadth-first search algorithm to mark the depths of all the signal lines in a hierarchical representation of the circuit. The primary outputs of the circuit are entered into a queue Q and their depths are marked

```
procedure  Mark_Depth-Hierarchical( )
begin
    Let M be the set of all modules in the netlist;
    Let L be the set of all lines in the netlist;
    Let PO be the set of all primary output lines;
    Q := NIL;          /* Queue for Breadth-first Search */

    for all lines l ∈ PO do begin
        l.depth = 1;
        enter l into queue Q;
    end
    while (Q ≠ NIL) do begin
        Delete entry l from Q;
        Let m be a module in M such that l is
            the output of m;
        Let I be the set of inputs to m;
        for each line i in I do begin
            Let F be the set of fanouts of line i;
            Let f be a fanout branch in F which feeds gate m;
            Let V be the set of virtual fanouts from fanout f to output l;
            for each virtual fanout v in V do begin
                v.depth := l.depth + delay(f_v_l);
                f.mark := f.mark + 1;
                if (f.mark = # virtual fanouts of f) then
                begin
                    f.depth := max( v.depth ) ∀ v ∈ V;
                    i.mark := i.mark + 1;
                end
            end
            if (i.mark = number of fanout branches of i) then
            begin
                i.depth := max( f.depth ) for all f in F;
                if i is not a primary input then
                    enter i into queue Q;
            end
        end
    end
end
```

Fig. 3. Procedure *Mark_Depth* for a hierarchical netlist. The notation f_v_l denotes a module-internal path from f to l through v.

as 1. The queue Q is used by the breadth-first search algorithm to traverse the signal lines in the circuit. An entry l is deleted from the queue Q and the module m, whose output l is, is identified. The inputs to the module m are examined one by one. An input i to module m may have a number of physical fanout stems, and each fanout branch may be associated with a set of virtual fanouts. The delay of the module-internal path from a virtual fanout branch v of the physical fanout branch f to the output l is computed using the procedure *delay*. The depth of the virtual fanout branch v is computed as the sum of the path delay above and the depth of the primary output l. The depth of the physical fanout branch f is computed as the maximum of the depths of all virtual fanout branches associated with f. Finally, the depth of the input i is the maximum of the depths of all fanout branches associated with i. The above computation is repeated for all signal lines of the circuit traversing the circuit from the primary outputs of the circuit towards the primary inputs of the circuit in a breadth-first-search manner. The procedure *Mark_Level_Hierarchical* is similar to the procedure for marking depths, except that the circuit is traversed from the primary input side to the primary output side in a breadth-first search manner.

Now we extend Theorem 1 for path selection in module-level circuits.

Theorem 2 (Hierarchical longest path theorem). *In a modular circuit, the minimum set of paths from primary inputs to primary outputs such that each circuit lead is included in at least one path whose propagation delay is no less than the delay of any path containing the lead comprises the longest path through* (a) *all primary*

inputs (b) *all fanout branches except the ones having the highest depth* (*with respect to a primary output*) *at their fanout stem*, *and* (c) *all virtual fanout branches except the ones having the highest depth at their virtual fanout stem.*

The algorithm for path generation in a modular circuit based on the hierarchical longest path theorem consists of three phases and is described in Figs. 5, 6, and 7. These subprocedures correspond to criteria (a), (b) and (c) of the hierarchical longest path theorem respectively. The procedure *Hierarchical_Path_Select_Phase*1 identifies the set of longest paths from each primary input to a primary output. For a given primary input l, the **while** loop in the procedure identifies a longest path from l to a primary output by traversing forward in the circuit from l to signals that are reachable from l until a primary output is encountered. During this forward traversal, the *depths* of the signal lines computed using the *Mark_Depth* procedure are utilized. For a line l, we identify a fanout branch $fmax$ such that $fmax$ the the highest depth among all the fanout branches of l. Similarly, we identify the virtual fanout branch $vmax$ such that its depth is highest among the virtual fanout branches associated with $fmax$. The path from l to $vmax$ is concatenated to the longest path identified thus far. The **while** loop terminates when a primary output is encountered during the forward traversal.

```
procedure Mark_Level-Hierarchical( )
begin
    Let M be the set of all modules in the netlist;
    Let L be the set of all lines in the netlist;
    Let PI be the set of all primary inputs;
    Q := NIL;

    for all lines l ∈ PI do begin
        l.level = 1;
        enter l into queue Q;
    end
    while (Q ≠ NIL) do begin
        delete entry l from Q;
        Let F be set of all fanout branches of line l;
        for each fanout f in F do begin
            Determine m in M which has f as an input;
            m.mark := m.mark +1;
            Let V be the set of all virtual fanouts from
                fanout f of line l at the input of module m;
            for each virtual fanout v in V do
                v.level := l.level + delay( v );
            if (m.mark = # of inputs to m) then begin
                Let O be set of all output lines of module m;
                for each output line o in O do begin
                    Let U be the set of all virtual fanouts
                    ending at output o of m
                    o.level = max(u.depth) ∀ u ∈ U;
                    if o is not a primary output then
                        enter o into queue Q;
                end
            end
        end
    end
end
```

Fig. 4. Procedure *Mark_Level* for a hierarchical netlist.

```
procedure Hierarchical_Path_Select_Phase1( )
begin
    Let M be the set of all module in the netlist;
    Let L be the set of all lines in the netlist;
    Let PI be the set of all the primary inputs;

    for each line l in PI do begin
        path := NULL;            /* Initialize a path */
        while (l is not a primary output) do begin
            Let F be the set of fanouts of line l;
            fmax = f such that f.depth is maximum ∀ f ∈ F;
            Let m be the module in M such that
                    fmax is an input to m;
            Let V be the set of all virtual fanouts
              from fanout f of l at the input of module m;
            vmax = v such that v.depth is maximum ∀ v ∈ V;
            path := concatenate ( path , l_vmax );
            Determine l such that l is the output line of m
                    which is fed by vmax;
        end
        print path;
    end
end
```

Fig. 5. Longest paths from primary inputs in a hierarchical netlist.

```
procedure Hierarchical_Path_Select_Phase2( )
begin
  for each line l in L do begin
    Let E be the set of all fanout branches of
      line l except fmax;
    for each fanout e in E do begin
      path := NULL;
      Let m be the module fed by fanout e of line l;
      Let V be the set of all virtual fanouts from fanout f
                        of line l at the input of module m;
      vmax = v such that v.depth is maximum ∀ v ∈ V;
      path := concatenate( path , l_vmax );
      Determine o such that o is the output line of m fed by vmax;
      while (o is not a primary output) do begin
        Let F be the set of fanouts of line o;
        fmax = f such that f.depth is maximum ∀ f ∈ F;
        Let m be the module in M such that fmax is an input to m;
        Let V be the set of all virtual fanouts
        from fanout f of line o at the input of module m;
        vmax = v such that v.depth is maximum ∀ v ∈ V;
        path := concatenate( path , o_vmax );
        Determine o such that o is the output line of m fed by vmax;
      end
      Let m be a module in M such that l is the output of m;
      Let I be the set of input lines to m;
      imax = i such that i.level is maximum ∀ i ∈ I;
      while ( imax is not a primary input ) do begin
        Determine v such that v is the virtual fanout
            with the longest delay
            from imax to module output;
        path := concatenate( imax_v, path );
        Let m be a module in M such that imax is the output of m;
        Let I be the set of input lines to m;
        imax = i such that i.level is maximum ∀ i ∈ I;
      end
      print path;
    end
  end
end
```

Fig. 6. Longest paths corresponding to criterion (b) of Theorem 2.

Procedure *Hierarchical_Path_Select_Phase*2 of Fig. 6 applies Rule (b) of the hierarchical longest path theorem. For every signal line l in the circuit, the procedure considers all the fanout branches except those which correspond to $fmax$, i.e., a branch whose stem has the highest level with respect to a primary output. Note that $fmax$ have been identified by the procedure *Hierarchical_Path_Select_Phase1*. For each selected fanout branch e, a longest path from a primary input i to a primary output o is identified using the two **while** loops in the procedure *Hierarchical_Path_Select_Phase*2. The first loop traverses from branch e forward towards the primary outputs and the second loop traverses from e backward towards the primary inputs. In moving forward, the first **while** loop maximizes the signal *depth* at each stage, whereas the second **while** loop maximizes the signal level at each stage. This ensures that the path identified for each fanout branch e is a longest path. The procedure *Hierarchical_Path_Select_Phase*2 of Fig. 7 applies Rule (c) of the hierarchical longest path theorem. The procedure is similar in spirit to *Hierarchical_Path_Select_Phase*2.

The performance of hierarchical path selection algorithms on several of module-level circuits will be described in Section 4.

3. Test Generation

MODET (MOdular DElay Tester) is a PODEM-based [6] test generation procedure. PODEM is a automatic

```
procedure Hierarchical_Path_Select_Phase3( )
  for each line l in L do begin
    Let F be the set of all fanout branches of line l;
    for each fanout f in F do begin
      Let V be the set of all virtual fanouts of f;
      vmax = v such that v.depth is maximum ∀ v ∈ V;
      Let U ⊂ V such that U contains all v ∈ V except vmax;
      for each virtual fanout u in U do begin
        path := NULL;
        path := concatenate ( path , u )
        Let m be the module fed by fanout f of line l;
        Determine o such that o is the output line of m fed by u;
        while (o is not a primary output) do
        begin
            Let F be the set of fanouts of line o;
            fmax = f such that f.depth is maximum ∀ f ∈ F;
            Let m be the module in M such that fmax is an input to m;
            Let V be the set of all virtual fanouts from fanout f of line o
                at the input of module m;
            vmax = v such that v.depth is maximum ∀v ∈ V;
            path := concatenate( path , o_vmax );
            Determine o such that o is the output line of m fed by vmax;
        end
        Let m be a module in M such that l is the output of m;
        Let I be the set of input lines to m;
        imax = i such that i.level is maximum ∀ i ∈ I;
        while ( imax is not a primary input ) do
        begin
            Determine v such that v is the virtual fanout with the longest delay
                from imax to module output;
            path := concatenate( imax_v , path );
            Let m be a module in M such that imax is the output of m;
            Let I be the set of input lines to m;
            imax = i such that i.level is maximum ∀ i ∈ I;
        end
        print path;
      end
    end
end
```

Fig. 7. Longest paths corresponding to criterion (c) of Theorem 2.

test pattern generator for combinational logic circuits for single stuck-at faults; it is based on the paradigm of branch-and-bound. PODEM implicitly examines all possible combinations of primary inputs as possible test vectors for a given fault. A high level description of **MODET** is given as a flowchart in Fig. 8.

3.1. Determine Objectives and Prioritize them

Given a path delay fault $\langle p, type\rangle$, delay test generation attempts to propagate the desired transition type from the primary input in which the path p is rooted to the primary output where p terminates. Such a propagation requires that proper logic values be assigned to the off-path lines of the modules on path p. This is called *off-path sensitization*, and is determined by the precomputed test vectors for the module types stored in module library. Off-path sensitization results in a list of *objectives* to be achieved, where an objective (l, v) is the assignment of a logic value v to an off-path line l in the circuit. Consider generating a test for the fault $\langle a(1)\ f(2)\ i(0), U1\rangle$ in the 2-bit ripple carry adder of Fig. 9. The primary objective is to propagate a rising transition ($U1$) from primary input $a(1)$[1] to line f and from line $f(2)$ to primary output $i(0)$.

The gate-level implementation of the full adder in our library is shown in Fig. 10. The precomputed library information for the full adder is summarized in Table 1. The gate-level path $\langle 174\rangle$ corresponds to the

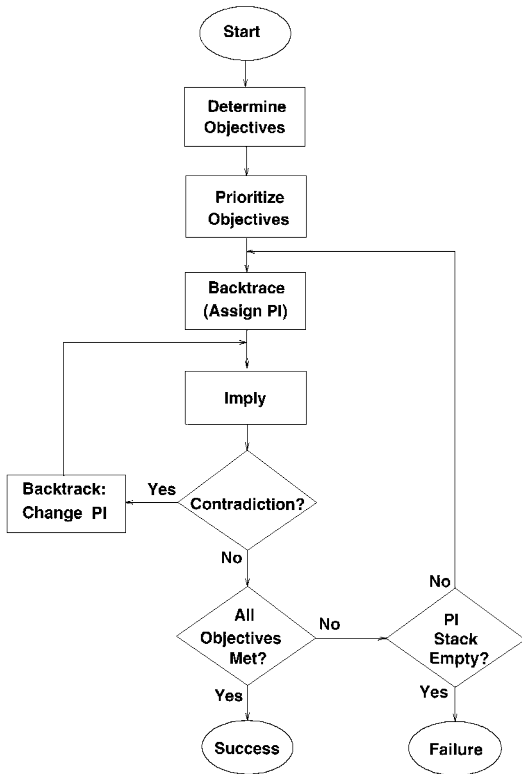

Fig. 8. **MODET** algorithm.

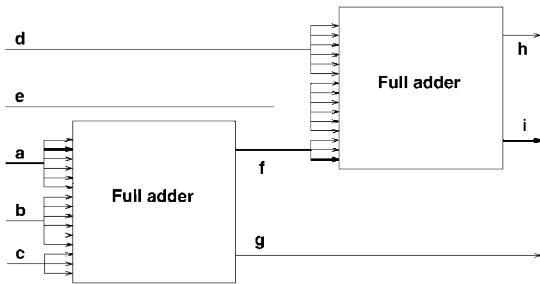

Fig. 9. Determining objectives for test generation.

Table 1. Library information for full adder.

Gate level path	Hierarchical path	Parity of path	Test vectors (↓)	Test vectors (↑)
1 7 4	1_0 4_0	even	U0 S1 U0	U1 U1 S0
1 8 12 4	1_1 4_0	even	U0 U0 S1	U1 S0 U1
1 6 9 10 5	1_2 5_0	even	U0 S0 S0	U1 S0 S0
1 7 9 11 5	1_3 5_0	even	U0 S1 S1	U1 S1 S1
0 1 6 9 11 5	1_4 5_0	odd	U0 S0 S1	U1 S0 S1
1 7 9 10 5	1_5 5_0	odd	U0 S1 S0	U1 S1 S0
2 7 4	2_0 4_0	even	S1 U0 U0	U1 U1 S0
2 8 12 4	2_1 4_0	even	U0 U0 S1	S0 U1 U1
2 6 9 10 5	2_2 5_0	even	S0 U0 S0	S0 U1 S0
2 7 9 11 5	2_3 5_0	even	S1 U0 S1	S1 U1 S1
2 6 9 11 5	2_4 5_0	odd	S0 U0 S1	S0 U1 S1
2 7 9 10 5	2_5 5_0	odd	S1 U0 S0	S1 U1 S0
3 12 4	3_0 4_0	even	S1 U0 U0	S0 U1 U1
			U0 S1 U0	U1 S0 U1
3 10 5	3_1 5_0	even	S0 S0 U0	S0 S0 U1
			S1 S1 U0	S1 S1 U1
3 11 5	3_2 5_0	odd	S0 S1 U0	S0 S1 U1
			S1 S0 U0	S1 S0 U1

hierarchical path $\langle 1_04_0 \rangle$; the notation a_j in a hierarchical path denotes the jth fanout of line a. The parity of a path denotes the number of inversions along the path. The test vector $\langle U0\ U0\ S1 \rangle$ is a robust test for the falling transition delay fault on the path $\langle 1_1\ 4_0 \rangle$. Similarly, $\langle U1\ S0\ U1 \rangle$ is a robust delay test for the rising transition delay fault on the path $\langle 1_1\ 4_0 \rangle$.

The hierarchical path $\langle a(1)\ f \rangle$ in the circuit of Fig. 9 corresponds to the hierarchical path $\langle 1_1\ 4_0 \rangle$ in the full adder module. Since the path delay fault on $\langle a(1)\ f \rangle$ is a rising path delay fault, we look up the robust test for a rising path delay fault on $\langle 1_1\ 4_0 \rangle$ from Table 1. This test vector happens to be $\langle U1\ S0\ U1 \rangle$. Further noting that the parity of the path $1_1\ 4_0 \rangle$ is even, we must set b to $S0$ and c to $U1$ to propagate the rising transition to f. This leads us to the objective $(b, S0)$

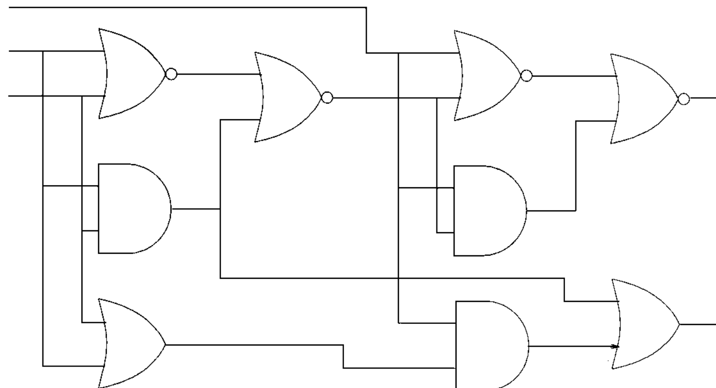

Fig. 10. Gate level diagram of a full adder.

and $(c, U1)$. Similarly, the reader may verify that the objectives $(d, S0)$ and $(e, S1)$ are generated by the path delay fault $\langle f(2)\ i\rangle\langle U1\rangle$.

Thus the precomputed tests for the path delay fault $\langle a(1)\ f, U1\rangle$ and $\langle f(2)\ i, U1\rangle$, stored as part of the library information for a full adder leads to the following objectives—$(b, S0)$, $(c, U1)$, $(d, S0)$, $(e, S1)$.

From this list of objectives, one which is not already satisfied is chosen based on the following heuristics aimed at reducing the number of backtracks [5].

1. Order the objective list such that the entries of the form $(l, S1)$ or $(l, S0)$ come before entries of the form $(l, U1)$ or $(l, U0)$.
2. Further order the objective list in the descending order of the degree of difficulty in satisfying an objective. A suitable measure for the degree of difficulty of an objective (l, v) is the minimum distance of line l from one of the primary inputs, signifying that closer a line is to the primary inputs, the easier it is to satisfy the objective.

An unsatisfied objective (l, v) is selected from the prioritized objective list, and, to satisfy the objective, an assignment of a value V is attempted to a *primary input line* I. We now explain how I and V are selected using the procedure *Back_trace* (Fig. 11). A backward search is made from line l towards the primary inputs of the circuit; when we backtrace from an output line of a module M, we may have more than one input of M which is yet unassigned. We choose the input line of M which is closest to the primary inputs, i.e., whose distance from the primary inputs is minimum, with the belief that it will be easier to satisfy the objective due to its proximity to primary inputs. The value V to be assigned to I is determined from the parity of the path from l to I; if the parity is odd, we set $V = \overline{v}$, else we set $V = v$.

After value V is assigned to a primary input I, the implications of the assignment are determined. If there is a contradiction between any of the objectives in the objective list and the implications of assignment $I = V$, the procedure *Back_Track* of Fig. 12 is invoked. When backtracking, I is set to an alternate value, say V', according to the following procedure. If $V \in S1, S0$, then $V' \in S1, S0$, and, similarly, if $V \in U1, U0$, then $V' \in U1, U0$. If both these assignments have been tried previously, then $V' = XX$. Thus we need not try all four values from the set $S1, S0, U1, U0$ at each primary input during Back_Track. This reduces the search space from 4^n to 2^n, where n is the number of primary inputs (see [16]). If the number of back tracks made during test generation for a fault f exceeds a predefined limit, we conclude that f is hard-to-test and terminate test generation.

4. Experimental Results

MODET was implemented on a Sun-SPARC workstation (16MB memory, 80 MHz speed) using about

```
procedure Back_Trace( )
begin
        Let M be the set of all module in the netlist;
        Let L be the set of all lines in the netlist;
        Let PI be the set of all primary inputs;
        Let O be the list of objectives;
        Let o ∈ O be the initial objective;
        Let l ∈ L be the objective net and v be the
                  objective value of o;
    while ( l is not a primary input ) do begin
        Let m be a module in M such that l is the output of m;
      Let I be the set of input lines to m;
                  if (any input i ∈ I has value XX) then
                  begin
                  l = i such that i.level is minimum ∀ i ∈ I that are unassigned;
                          if (internal path of m corresponding to
                                  minimum level has odd parity) then
                          begin
                                  v = not(v);
                          end
                  end
        end
end
```

Fig. 11. Procedure *Back_Trace*.

```
procedure Back_Track( )
begin
    Let L be the set of all lines in the netlist;
    Let PI be the set of all primary inputs;
    Let S be the stack of all p ∈ PI that have been
    assigned during procedure back_trace;
    while (S ≠ NULL) do begin
    pop entry p from S;
        if (all alternatives have not been tried at p) then begin
            make an alternative assignment to p;
            push p back into S;
            break;
        end
        Assign XX to p;
        Carry out the implications of assignment done above;
    end
end
```

Fig. 12 Procedure *Back_Track* used in **MODET**.

4000 lines of C code. In this section, we report experimental results obtained using **MODET** to bring out the performance of our tool. Although a number of benchmark circuits are available for delay test generation at gate-level, none are available at module level. We therefore used test circuits of our own, such as ripple carry adders, parity checker trees, comparators, N-to-1 multiplexers, etc.

To study the performance of hierarchical path selection, we experimented with a large number of gate-level and module-level circuits. Tables 2 and 3 in Section 4 confirm that the size of the selected set of paths grows more or less linearly with circuit size as compared to the exponential increase of the total number of paths in a circuit. The hierarchical path selection algorithm requires lesser CPU-time for modular circuits than for gate-level circuits. The savings become more significant with increase in circuit size and level of abstraction. On the other hand, the approach of flattening the circuit and performing the path selection at gate-level

Table 2. Path selection results for n-bit ripple carry adders.

	Modules		Nets		Sel. paths				Time (s)		
n	HL	GL	HL	GL	HL	GL	Mapped	Tot. paths	HL	GL	Mapped
2	10	26	9	23	21	21	21	37	0.017	0.021	0.054
4	18	50	17	45	41	41	41	105	0.022	0.035	0.068
8	34	98	33	89	81	81	81	337	0.034	0.066	0.129
16	66	194	65	177	161	161	161	1185	0.062	0.166	0.302
32	130	386	129	353	321	321	321	4417	0.170	0.579	0.983
64	258	770	257	705	641	641	641	17025	0.632	2.501	3.807

Table 3. Path selection results for n-bit parity checkers.

	Modules		Nets		Sel. paths				Time (s)		
n	HL	GL	HL	GL	HL	GL	Mapped	Tot. paths	HL	GL	Mapped
2	4	6	3	5	4	4	4	4	0.012	0.014	0.034
4	8	14	7	13	10	10	10	16	0.013	0.018	0.041
8	16	30	15	29	22	22	22	64	0.021	0.022	0.060
16	32	62	31	61	46	46	46	256	0.027	0.036	0.102
32	64	126	63	125	94	94	94	1024	0.042	0.080	

requires more CPU-time due to the flattening and mapping overheads.

Table 2 shows the results of running our path selection algorithms on an n-bit Ripple Carry Adder (RCA), $2 \leq n \leq 64$. For both gate-level (GL) and high-level (HL) circuits, we show the number of modules in the netlist, the number of nets, the number of paths selected, and the execution time for running the path selection algorithms. In the table, the number of selected paths are reported for both the selection approaches discussed in Section 2. Results for Approach 1 are given under column *Mapped* and for Approach 2 under column HL. We observe that the number of selected paths grows more or less linearly with n, although the total number of paths increases sharply with n. The path selection algorithm (Approach 2) requires more CPU-time on gate-level circuits. Path selection using Approach 1 was found to be slower than gate-level path selection; this can be attributed to the overheads of flattening the circuit and mapping the gate-level path to a modular path. We can conclude that Approach 2 is superior to Approach 1 in terms of execution time, and the improvement becomes more and more visible with increase in circuit size. In fact, Approach 1 has larger memory requirements during the flattening phase; for a 32-bit parity checker circuit (see Table 3), Approach 1 failed due to the large memory requirement. The number of paths selected in the modular circuit was almost identical to that in the gate-level circuit. The savings in execution time become significant for larger values of n.

We conducted similar experiments for three other classes of circuits, namely, comparators, multiplexers, and parity checkers. To implement an n-bit comparator, we used a half adder, $n-1$ copies of full adders, inverters, 2-input OR and NOR gates. An n-input multiplexer can be constructed using $n-1$ 2-input multiplexers using a multiplexer tree construction. An n-bit parity checker can be made from 2-bit EXOR gates. The results obtained for all these classes of circuits were similar to those for RCA circuits.

Experiments on test generation were performed to compare the time taken and the number of back tracks required for test generation for a module level description as against gate-level. Table 4 shows the results of **MODET** on an n-bit ripple-carry adder, $2 \leq n \leq 64$. For each value of n, we show the number of delay path faults, the number of test vectors, the number of faults detected, the number of backtracks, and the test generation time. The fault set for each circuit was

Table 4. Test generation results for n-bit ripple carry adders.

		# vectors		# detected		# backtracks		Time (s)	
n	# faults	HL	GL	HL	GL	HL	GL	HL	GL
2	42	36	38	42	42	0	4	0.044	0.058
4	82	72	74	82	82	0	12	0.059	0.104
8	162	144	150	162	162	0	56	0.120	0.262
16	322	288	302	322	322	0	240	0.425	0.902
32	642	576	606	642	642	0	992	1.497	4.067
64	1282	1152		1282		0		7.734	

obtained by running the hierarchical path selection algorithm of Section 2 (Approach 2), and associating each type of transition, ↑ and ↓, with every selected path p. The backtrack limit in our experiments was set to 30. With this limit, we did not encounter any case of fault dropping. The number of backtracks in a gate-level delay test generation increases sharply with increase in circuit size. On the other hand, there were no backtracks in running **MODET** for an RCA composed of full adders as modules. This can be explained by observing that in a module-level RCA, the objectives involve either the primary inputs or the ripple carry lines. When primary inputs are involved as part of an objective, no backtrack or backtrace will be necessary. Even when objectives included ripple carry lines, we found that no backtracks were necessary to satisfy the objectives. We obtained 100% fault coverage for all the circuits. We can conclude from Table 4 that the test generation time is lower when **MODET** is used on hierarchical netlists. Using the UNIX program profiler, we analyzed that this saving in time was mainly due to faster logic simulation at hierarchical level. There is no entry in Table 5 for a 64-bit gate-level RCA since the test generator fails due to lack of memory.

Tables 4 and 6 show the results of **MODET** for n-bit parity checkers and multiplexer trees respectively. The

Table 5. Test generation results for n-bit parity checkers.

		# vectors		# detected		# backtracks		Time (s)	
n	# faults	HL	GL	HL	GL	HL	GL	HL	GL
2	8	8	8	8	8	0	0	0.032	0.037
4	20	20	20	20	20	4	3	0.038	0.044
8	44	44	44	44	44	12	12	0.054	0.075
16	92	92	92	92	92	28	33	0.119	0.213
32	188	188	188	188	188	60	78	0.401	0.801

Table 6. Test generation results for n-bit multiplexers.

		# vectors		# detected		# backtracks		Time (s)	
n	# faults	HL	GL	HL	GL	HL	GL	HL	GL
2	8	6	6	8	8	0	0	0.031	0.33
4	20	16	16	20	20	0	0	0.036	0.043
16	92	76	76	92	92	36	0	0.114	0.143
32	188	156	156	188	188	116	0	0.355	0.391

results are similar to those obtained for a ripple carry adder. In our experimentation on **MODET** using mux-trees, we observed that there were no backtracks in a gate-level multiplexer; whereas backtracks were necessary for test generation at a hierarchical level. Despite this fact, speedup in test generation was observed since 5-valued logic simulation is much faster for a hierarchical circuit.

5. Conclusion

We have presented a scheme for the generation of delay tests in circuits described at the module level. This scheme, called **MODET**, includes software tools for Path Selection and Automatic Test Generation. **MODET** is also applicable to mixed-level circuits; it considers a path delay fault model at module-level circuits. We have described novel techniques for module-level path selection and test generation. A major feature of **MODET** which contributes to its impressive performance is the extensive use it makes of library information of the modules handled by it. We have tested our tools on several circuits at various levels of abstraction and found that in each case there were large gains in terms of CPU time and memory requirements for hierarchical circuits as compared to their gate-level implementations.

Acknowledgments

We express our gratitude to Prof. S.M. Reddy of University of Iowa for his constant encouragement and helpful suggestions during this work. Discussions with Ajay Mittal and Gurjeet Saund were very useful. We are also thankful to the anonymous referees and the editor of JETTA for their comments which helped us in improving the original draft of this paper.

Note

1. $a(1)$ refers to the second virtual fanout of line a.

References

1. G.L. Smith, “Model for Delay Faults Based on Paths,” *Proceedings of IEEE International Test Conference*, 1985, pp. 342–349.
2. K.D. Wagner, “The Error Latency of Delay Faults in Combinational and Sequential Circuits,” *Proceedings of IEEE International Test Conference*, 1985, pp. 334–341.
3. C.J. Lin and S.M. Reddy, “On Delay Fault Testing in Logic Circuits,” *IEEE Transactions on CAD*, Vol. CAD-6, No. 5, pp. 694–703, Sept. 1987.
4. A.K. Majhi, J. Jacob, L.M. Patnaik, and V.D. Agrawal, “An Efficient Automatic Test Generation System for Path Delay Faults in Combinational Circuits,” *Proceedings of the 8th International Conference on VLSI Design*, 1995, pp. 161–165.
5. S. Patil and S.M. Reddy, “A Test Generation System for Path Delay Faults,” *Proceedings of International Conference on Computer-Aided Design*, 1989, pp. 40–43.
6. P. Goel, “An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits,” *IEEE Transactions on Computers*, Vol. C-30, No. 3, pp. 215–222, March 1981.
7. K. Heragu, V. D. Agrawal, and M.L. Bushnell, “Statistical Methods for Delay Fault Coverage Analysis,” *Proceedings of the 8th International Conference on VLSI Design*, 1995, pp. 166–170.
8. W.N. Li, S.M. Reddy, and S. Sahini, “On Path Selection in Combinational Logic Circuits,” *Proceedings of the 25th ACM/IEEE Design Automation Conference*, 1988, pp. 142–147.
9. D. Bhattacharya, P. Agrawal, and V.D. Agrawal, “Test Generation for Path Delay Faults Using Binary Decision Diagrams,” *IEEE Transactions on Computers*, Vol. 44, No. 3, pp. 434–447, March 1995.
10. M. Abromovici, M.A. Breuer, and A.D. Friedman, *Digital Systems Testing and Testable Design*, W.H. Freeman, 1990.
11. J.D. Calhoun and F. Brglez, “A Framework and Method for Hierarchical Test Generation,” *IEEE Transactions on Computer-Aided Design*, Vol. 11, No. 1, pp. 45–67, Jan. 1992.
12. S.J. Chandra and J.H. Patel, “A Hierarchical Approach to Test Vector Generation,” *Proceedings of the 24th IEEE/ACM Design Automation Conference*, 1987, pp. 495–501.
13. T.M. Sarfert, R.G. Markgraf, M.H. Schulz, and E. Trischler, “A Hierarchical Test Pattern Generation System Based on High-level Primitives,” *IEEE Transactions on Computer-Aided Design*, Vol. 11, No. 1, pp. 34–44, Jan. 1992.
14. I. Pomeranz and S.M. Reddy, “On Testing Delay Faults in Macro-based Combinational Circuits,” *Proceedings of the 30th ACM/IEEE Design Automation Conference*, 1994, pp. 332–339.
15. G.S. Saund, N. Agrawal, and P. Agarwal, Hierarchical Testing and Testability Analysis, Master’s Thesis, Electrical Engineering, IIT Delhi, India, May 1995.
16. S. Patil, An Automatic Test Pattern Generator for Delay Faults in Logic Circuits, Master’s Thesis, Electrical and Computer Engineering, University of Iowa, May 1987.

C.P. Ravikumar obtained his Ph.D. in Computer Engineering from the Department of Electrical Engineering Systems, University of

Southern California (1991). He received a Master of Engineering degree in Computer Science from the Department of Computer Science and Automation, Indian Institute of Science (1987). He obtained a Bachelor of Engineering degree in Electronics from Bangalore University (1983). During 1991–1995, he served the Department of Electrical Engineering, Indian Institute of Technology, Delhi as an Assistant Professor. Since 1995, he is an Associate Professor in the Department of Electrical Engineering, IIT Delhi. He is the author of the book Parallel Algorithms for VLSI Layout Design published by Ablex Publishers, New Jersey (1996). He is the Indian editor of the International Journal of VLSI Design published by Gordon and Breach. He serves on the editorial committee of the journal "Computers and Informatics" published by the Computer Society of India. His research interests are in the areas of high-level synthesis and testing of VLSI circuits and high-performance computing.

Nitin Agrawal obtained a Bachelor of Technology degree from the Department of Electrical Engineering, IIT Delhi in 1995. He is currently a member of the technical staff at S3 India, where he works as a computer architect and a VLSI design engineer. His research interests are in the areas of hierarchical testing and fault analysis, and high-performance computer architecture.

Parul Agarwal was born in Delhi, India, on December 9, 1973. She received a Bachelor of Technology degree in Electrical Engineering from the Indian Institute of Technology, Delhi in 1995. She is currently doing her M.S. in computer science in University of Michigan, Ann Arbor, Michigan. Her research interests include VLSI design and testing, VLSI CAD, distributed systems and hardware/software codesign. She is a recipient of the Rackham fellowship for graduate studies at University of Michigan. She was awarded merit prizes at the Indian Institute of Technology for standing in the top 2 percent of the class. She also got selected for an exchange program to University of Masachussetts, Amherst in 1993 and is a recipient of the prestigious National Talent Search scholarship which is awarded to less than 0.25% of 3,00,000 applicants nationwide.