An Efficient Method for Computing Exact Path Delay Fault Coverage

Bhanu Kapoor
Integrated Systems Laboratory, Texas Instruments, Inc.
P. O. Box 655474, MS 446, Dallas, TX 75265
email: kapoor@hc.ti.com

Abstract

We describe algorithms and data structures for accurate and efficient computation of path delay fault coverage. Our method uses an interval-based representation of consecutively numbered path delay faults. We describe a modified 2-3 tree data structure to store and manipulate these intervals to keep track of tested faults. Some results obtained using non-robust simulation of benchmark circuits suggest the viability of this approach.

I. Introduction

The maximum allowable path delay in a synchronous computer is determined by its clock rate. If the delay on a path of a manufactured circuit exceeds the time period between clocks, incorrect outputs may be latched. The objective of delay testing is the ensure that the manufactured network operates correctly at the functional clock rate.

Two models have been proposed for delay faults: the gate delay fault [2] and the path delay fault [15]. According to gate delay fault model, any input to a gate can be subjected to a delay fault that may cause the response of the gate to be slow compared to its specification. The path delay fault model focuses on the aggregate delay along the path and not on the individual delays of the gates comprising the paths.

The path delay fault model is capable modeling distributed failures resulting due to statistical variation in the manufacturing process. As a result, it is of particular importance for circuits designed using statistical timing analyzers. This model has been researched extensively for the tasks of test generation [4, 8, 9, 10, 13], fault simulation [3, 7, 11, 14, 15], and synthesis for testability [5, 6, 16].

One problem associated with path delay fault model is that there can be an exponential number of path delay faults in a circuit. This makes it practically impossible to enumerate all paths for the purpose of test generation and fault simulation. The problem of path delay fault simulation was investigated in [15]. In order to compute the exact path delay fault coverage by a given test set, this methods have a worst case exponential complexity because complete set of paths is used for the purpose of fault coverage computation. In [14], all delay tested paths are stored, resulting in same complexity as that of the previous method.

A non-enumerative method to compute the path delay fault coverage is presented in [11]. A polynomial complexity algorithm is presented, however, it may compute a pessimistic estimate for the coverage. As the degree of the polynomial is increased, better estimates are obtained. Only when the degree of the polynomial is allowed to be of the order of the number of lines in the circuit, thus resulting in exponential complexity, the exact path delay fault coverage is obtained.

Symbolic representation of path delay-faults can achieve high degree of compaction [7] relative to more explicit forms. Large number of path delay-faults exist in common digital circuits. Ordered binary decision diagrams (OBDDs) provide a convenient data structure to represent these large number of path delay-faults during the process of fault simulation computing the path delay-fault coverage for a given delay test-set. The complexity in this case dependent upon the size of the OBDD constructed during the course of simulation.

In this paper, an efficient algorithm to compute exact path delay fault coverage has been described. It is based on the following key observations: (1) The number of paths in the circuit can be computed in time which is linear in the number of lines in the circuit, by making one pass over the circuit. (2) A set of consecutively numbered path delay faults can be represented as a closed interval over a pair of integers. A tree-based data structure can be used then to store and manipulate these intervals [12]. In addition to estimating exact path delay fault coverage, this technique also provides the ability to efficiently find out whether a given path delay fault has been tested. This can be used to guide a delay test generator to use its resources on untested path delay faults only.

The paper is organized as follows: After the preliminaries of Section 2, the paper presents algorithms and data structures in Section 3. Section 4 contains some discussion on the experimental results. We conclude the paper with a summary.

II. Preliminaries

A combinational circuit is represented as a directed acyclic graph (DAG), $G(V, E)$, where gates correspond to nodes and the wires correspond to edges in the graph. Fig. 1(a) shows a combinational circuit which is modeled as a directed acyclic graph in Fig. 1(b).

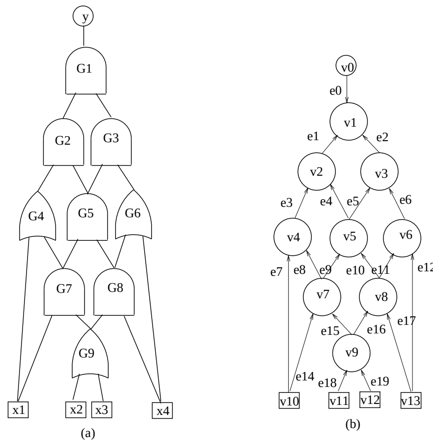

Figure 1: DAG representation of path delay faults

The fault simulator described here incorporates models for *robust* [15] and *non-robust* [8, 10] tests. A two-pattern test $< V_1, V_2 >$ is called a *robust test* for a fault on a path P, if it detects that fault independently of all other delays in the circuit. A two-pattern test $< V_1, V_2 >$ is called *non-robust* test for a fault on path P, if it detects fault under the assumption that the off-path sensitizing inputs of all gates on the structural path corresponding to the P stabilize at their final values prior to the time at which the transition along P arrives at them.

For circuit with large number of paths, an explicit fault representation is not practical. Efficient path manipulation algorithms are necessary for execution speed as well as to maintain manageable memory requirement. An efficient path representation scheme has been presented in [3]. This scheme allows for sequential numbering of the paths in a circuit. This is achieved through a combination of a linear algorithm for path counting and structural ordering of paths through each node. For the work presented here, the two path faults, rising and falling, associated with path i are numbered $2i - 1$ and $2i$, respectively.

III. Algorithms and Data Structures

The method described in this paper simply uses the fact that a set of consecutively numbered path delay faults i_1, $i_1 + 1$, ... ,i_2 can be represented as a closed interval $[i_1, i_2]$, with $i_1 \leq i_2$. The closed interval $[i_1, i_2]$ represents a set of $i_2 - i_1 + 1$ path delay faults. During the course of path delay fault simulation, newly tested path delay faults, represented as intervals, are merged into existing intervals. This allows for a memory efficient representation of path delay faults.

3.1 Interval-Based 2-3 Tree

We represent an interval $[i_1, i_2]$ as an object x, with fields $low[x] = i_1$ and $high[x] = i_2$. We say that intervals x and y *overlap* if $x \cap y \neq \phi$, that is, if $low[x] \leq high[y]$ and $low[y] \leq high[x]$. We say that two intervals x and y are *adjacent* if either $low[x] = high[y] + 1$ or $low[y] = high[x] + 1$. An *interval tree* maintains a dynamic set of elements, with each element containing an interval x. We have used the 2-3 tree data structure [1] to store intervals.

We now define the interval-based 2-3 tree data structure. It has the following properties:

(1) Each interior node has two or three children.

(2) Each path from root to leaf has the same length.

(3) Each leaf contains an interval.

(4) The elements in the tree are ordered by a linear order. The linear ordering $\prec$ is defined as follows: If there exist two intervals x and y then $x \prec y$ if and only if $low[x] < low[y]$.

(5) At each interior node we record two intervals. First interval, denoted by R, contains the smallest and the largest numbers stored in the tree rooted at that node. The second interval, denoted by S, consists of the smallest number descending from the second child and, if there is a third child, we also record the smallest number descending from the third child. At each node, we also record the total number of tested faults, N_v, stored in the tree rooted at that node.

The interval-based 2-3 tree data structure requires at least $1 + log_3 n$ levels and no more than $1 + log_2 n$ levels. Thus, the path lengths in the tree are $O(logn)$. An example of such a tree is shown in Fig. 2. Each leaf is a set of path delay faults represented as an interval. Each interior node contains the information about the sub-tree rooted at that node.

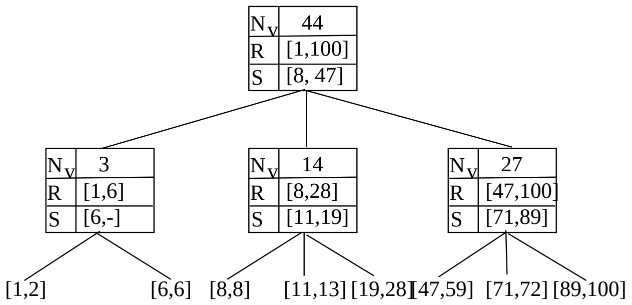

Figure 2: An interval-based 2-3 tree

Our interval tree supports the following operations:

`INTERVAL-INSERT` $(T,\ x)$ adds the interval x to the interval tree T.

`INTERVAL-DELETE` $(T,\ x)$ removes the interval x from interval tree T.

`INTERVAL-SEARCH` $(T,\ x)$ returns an element y in the interval tree such that y either overlaps or is adjacent to interval x. If there is no such interval then it returns a NIL.

We define a merge operation over intervals as follows: INTERVAL-MERGE (x, y) returns an interval z such that $low[z] = min(low[x], low[y])$ and $high[z] = max(high[x], high[y])$.

INTERVAL-INSERT and INTERVAL-DELETE are similar to the insert and delete operations over 2-3 trees. The key for an interval x is the low endpoint, $low[x]$. An inorder tree walk of the data structure lists the intervals in sorted order by low endpoint. Some examples of insertion in the interval tree are as follows:

Consider insertion of an interval x = [4,4] in the interval tree shown in Fig. 2. Since $low[x] < low[S]$, we move to the first child of the root node. At this node also, we find that $low[x] < low[S]$. Since the current node is just one level above the leafs and contains only two children, interval x is inserted as the second child of the node. The R, S, and N_v fields of the intermediate nodes on the path between the newly inserted leaf and the root node are modified appropriately. The resulting interval tree is shown in Fig. 3.

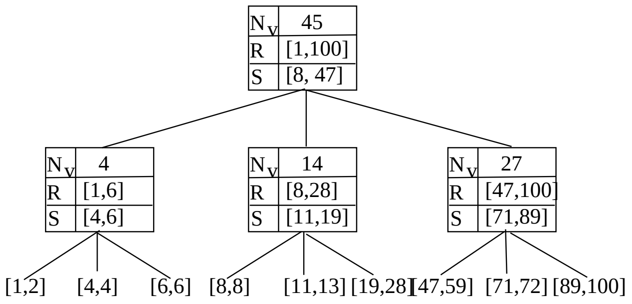

Figure 3: Insertion of [4,4] in the tree of Fig. 3

It is quite possible that the interval being inserted finds its place as a child of a node v which already contains three children. In this case, the node v is split into two nodes v and $\hat{v}$. The two intervals with smaller keys stay with v, while the two intervals with larger keys become children of $\hat{v}$. Now, $\hat{v}$ is inserted among the children of p, the parent of node v. If p had two children, we make $\hat{v}$ the third and place it immediately to the right of v. If p had three children before $\hat{v}$ was created, we split p into p and $\hat{p}$, and so on.

For example, consider insertion of x = [16,17] into tree of Fig. 3. The intended parent node of x already has three children. We split the parent node into two nodes. However, once the node is split, the root of the tree now contains four children. We further split the root into two nodes, adding an extra level in the tree. The fields of the affected nodes is modified accordingly. The resulting interval tree is shown in Fig. 4.

Just as INTERVAL-INSERT works in a fashion similar to the insert operation over 2-3 trees, INTERVAL-DELETE is analogous to the delete operation over 2-3 trees. INTERVAL-SEARCH looks for an overlapping interval with the smallest low endpoint. If such an interval does not exist then it looks for an adjacent interval. If it cannot find any such interval then the function returns a NIL value, indicating a failure.

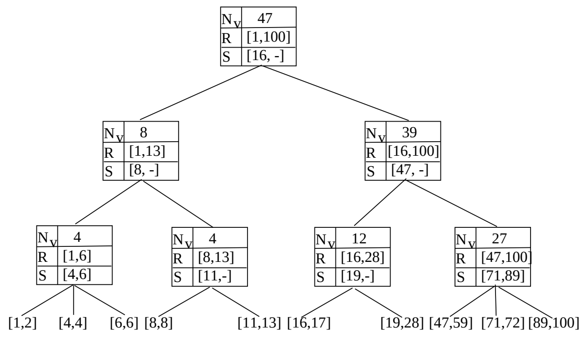

Figure 4: Insertion of [16,17] in the tree of Fig. 3

So far we have assumed that the interval being inserted does not overlap with any of the intervals already stored in the interval tree. When an interval x either overlaps with or is adjacent to some existing intervals, we will like to collapse these intervals into a single interval. This can be achieved using the following steps:

(1) We find all the overlapping and adjacent intervals using INTERVAL-SEARCH. As we find each such interval, it is stored in a list L and deleted from the tree T.

(2) The interval x and the intervals in the list L are merged to create a single interval y.

(3) The interval y is then inserted into the tree T.

Note that in the worst case, k deletions may be performed, where k is the number of overlapping and adjacent intervals. This gives an overall complexity of $O(k.logn)$, where n is the number of stored intervals in the tree.

So far, we have discussed the data structure and algorithms to maintain a set of closed intervals without telling how to compute these intervals during the course of path delay fault simulation. This is discussed next.

3.2 Computing Tested Faults as Intervals

We will like to to compute the set of tested path delay faults in terms of intervals. Once represented as a set of intervals, these intervals can be inserted in the interval tree. The root of the tree contains information about the total number of tested path delay faults at any stage of fault simulation. The task of representing newly tested path delay faults as a set intervals is accomplished by an algorithm as depicted in the following steps:

(1) We mark the edges of the tested path delay faults. Each primary input node is assigned an interval [1,1].

(2) The edges and nodes of the marked DAG are topologically sorted. The computation of intervals through each edge and node is carried out in the topologically sorted order.

(4) The set of intervals through a node is simply the union of intervals through the edges directed into that node. For example, if $[p, q]$ and $[m, n]$, with $p \leq m$, are the intervals

through the edges directed into node n_i then the the set of intervals through the node n_i is $([p, n])$ if $[p, q]$ and $[m, n]$ either overlap or are adjacent. Otherwise, it is $([p, q], [m, n])$.
(5) The set of intervals through an edge is propagated as follows: Let $([p1, q1], [p2, q2], ..., [pn, qn])$ be the set of path sequences through node n_i. Let node n_j be a parent node of n_i such that edge e_{ij} connects nodes n_j and n_i. Let us assume that node n_j has m edges directed into it, numbered 1 through m, and e_{ij} be the k^{th} edge. Then the set of intervals through the edge e_{ij} is given by $([p1 + N, q1 + N], [p2 + N, q2 + N], ..., [pn + N, qn + N])$, where N is the total number of path delay faults through first $k - 1$ edges directed into node n_j.

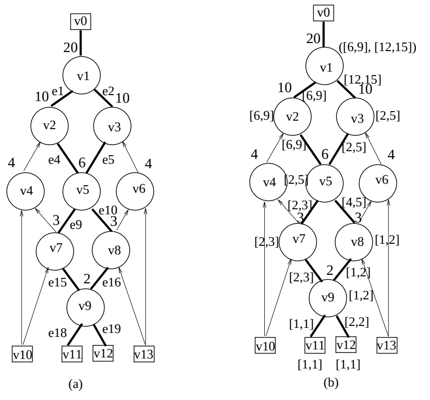

Figure 5: Computing tested faults as intervals

For example, consider the example shown in Fig. 5(a). The portion of the DAG shown using bold lines represents the tested path delay faults as a result of an application of a delay test. The computed set of path sequences is shown in Fig. 5(b). Let S(x) denote the set of path sequence for the edge or node denoted by x. The computation of tested path sequences, based on the algorithm described above, is carried out as follows :
S(v11) = ([1,1]), S(v12) = ([1,1])
S(e18) = ([1 + 0,1 + 0]) = ([1,1])
S(e19) = ([1+ 1,1 + 1]) = ([2,2])
S(v9) = S(e18) ∪ S(e19) = ([1,2])
S(e15) = ([1 + 1, 2 +1]) = ([2,3]), S(e16) = ([1,2])
S(v7) = ([2,3]), S(v8)= ([1,2])
S(e9) = ([2,3]), S(e10) = ([4,5])
S(v5) = S(e9) ∪ S(e10) = ([2,5])
S(e4) = ([6,9]), S(e5) = ([2, 5])
S(v2) = S(e4) = ([6,9]), S(v3) = S(e5) = ([2,5])
S(e1) = ([6,9]), S(e2) = ([2 + 10, 5 + 10]) = ([12,15])
S(v1) = S(e1) ∪ S(e2) = ([6,9],[12,15])

The set S(v1) contains the information that the set of tested path delay faults are the faults numbered 6, 7, 8, 9, 12, 13, 14, and 15. The complexity of the algorithm is $O(kE)$ where k is maximum number of intervals generated and E is the number of edges in the tested portion of the DAG.

IV. Experimental Results

The proposed algorithms for robust and non-robust detection of path delay faults has been implemented in LISP running on a SPARCStation 2. The results presented here are for non-robust path delay fault simulation of several benchmark circuits.

The simulation has been performed using randomly generated delay test vector pairs. For a circuit with n inputs, for each randomly generated n-bit binary vector, a set of $2n$ delay tests, corresponding to a rising and falling transition at each bit, were generated. Each circuit was simulated using 10,000 such delay test vectors. This does not put any restriction on the number of path delay faults to be considered.

In Table I, Column 2 indicates the number of possible path delay faults in each circuit. Column 3 contains the number of detected path delay faults as result of fault simulation using randomly generated 10,000 delay test vectors. The last column indicates the CPU times in seconds.

Column 4 contains the maximum cardinality of the set of intervals during the course of delay fault simulation. Typically this number is much smaller than the number of path delay faults detected. This is a result of merging consecutively numbered path delay faults into closed intervals. As more and more faults get tested, the number of intervals does not necessarily increase.

Table I: Results of non-robust fault simulation

Circuit Name	Path Faults	Faults Detec.	$\|S(x)\|$ max.	CPU
sn54181	914	802	24	12
apex6	2998	1630	126	86
des	199690	12574	843	187
rot	15886	1980	443	79
z4ml	182	182	12	9
c2670	13559768	2484	234	92
c3540	57353342	20172	1086	98
c5315	2682610	6912	1002	112
c6288	1.9788E20	116298	10451	132
c7552	1452988	7332	1345	154

As with all other known algorithm for computing accurate path delay fault coverage, our algorithm has a worst case performance that is unacceptable for all but the circuits with reasonable number of paths. It is easy to see this because a random case of simulation can result in testing all the odd-numbered or even-numbered path faults, resulting in a scenario that is no better than a direct method.

However, unlike other methods, all of which present unacceptable scenario for even the best case, our method provides hope for very efficient simulation. In the best scenario, it will require only $O(1)$ memory and time throughout the fault simulation process. The approach certainly seems to be efficient for random delay fault simulation, evident from results obtained over most of the examples considered in our experimentation.

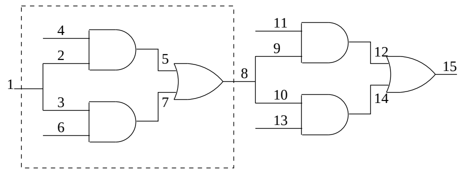

Figure 6: Example circuits with exponential number of detectable paths

Since the path delay fault testability of the ISCAS-85 and MCNC benchmark examples is relatively low, Pormeranz and Reddy [11] have constructed a set of circuits with large number of detectable path delay faults. The circuits have the structure shown in Fig. 6. The block in the dashed box is repeated different number of times, to obtain circuits of different sizes. In circuit C_n, the block is repeated n number of times. The number of path delay faults in these circuits is given by $3 \cdot 2^{n+1} - 4$. Experimental results for these circuits are given in Table II.

Table II: Results for C_n Circuits

Circuit	Faults Detected	CPU
C_{20}	6291452	1.6
C_{30}	6442450940	2.3
C_{40}	6597069766652	2.9
C_{50}	6755399441055740	3.8

V. Summary

We have presented a new algorithm for exact computation of path delay fault coverage for a given delay-test set. Our exact algorithm is based on the simple idea of representing consecutive path delay faults as an interval. We have described the algorithms and data structures to manipulate a set of intervals stored in a 2-3 tree. Some experimental results obtained using non-robust simulation of benchmark circuits suggest the viability and validity of our approach.

Acknowledgment

The author is grateful to Dr. J. Brock Barton of Texas Instruments and Dr. V. S. S. Nair of SMU, Dallas, for their valuable suggestions.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company, Reading, MA, 1983.

[2] Z. Barzilai and B. K. Rosen. Comparison of ac self-testing procedures. In *Proc. of International Test Conference*, pages 89–94, 1983.

[3] S. Bose, P. Agrawal, and V. D. Agrawal. Path delay fault simulation of sequential circuits. *IEEE Trans. on VLSI*, pages 453–461, December 1993.

[4] T. J. Chakraborty, V. D. Agarwal, and M. L. Bushnell. Delay fault models and test generation for random logic sequential circuits. In *Proc. of Design Automation Conference*, pages 165–172, 1992.

[5] S. Devadas and K. Keutzer. Synthesis of robust delay-fault-testable circuits: Practice. *IEEE Trans. on Computer-Aided Design*, pages 277–300, March 1992.

[6] S. Devadas and K. Keutzer. Synthesis of robust delay-fault-testable circuits: Theory. *IEEE Trans. on Computer-Aided Design*, pages 87–101, January 1992.

[7] Bhanu Kapoor. *Synthesis and Analysis of Delay Fault Testable Digital Circuits.* PhD thesis, Southern Methodist University, Dallas, TX, 1994.

[8] C. J. Lin and S. M. Reddy. On delay fault testing in logic circuit. *IEEE Trans. on Computer-Aided Design*, 6:694–703, September 1987.

[9] P. McGreer, A. Saldanha, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. Timing analysis and delay-fault test generation using path recursive functions. In *Proc. of International Conference on Computer-Aided Design*, pages 180–183, 1991.

[10] E. S. Park and M. R. Mercer. Robust and non-robust tests for path delay faults in combinational logic. In *Proc. of International Test Conference*, pages 1027–1034, 1987.

[11] I. Pomeranz and S. M. Reddy. An efficient non-enumerative method to estimate path delay fault coverage. In *Proc. of International Conference on Computer-Aided Design*, 1992.

[12] Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.

[13] J. Savir and W. H. McAnney. Random pattern testability of delay faults. *IEEE Trans. on Computers*, pages 291–300, March 1988.

[14] M. H. Schulz, F. Fink, and K. Fuchs. Parallel pattern fault simulation of path delay faults. In *Proc. of Design Automation Conference*, pages 357–363, 1989.

[15] G. L. Smith. A model for delay fault based on paths. In *Proc. of International Test Conference*, pages 342–349, 1985.

[16] T. W. Williams, B. Underwood, and M. R. Mercer. The interdepence between delay optimization of synthesized networks and and testing. In *Proc. of Design Automation Conference*, pages 87–92, 1991.