Static Logic Implication with Application to Redundancy Identification

Jian-Kun Zhao, Elizabeth M. Rudnick, and Janak H. Patel
Center for Reliable & High-Performance Computing
University of Illinois, Urbana, IL 61801

Abstract

This paper presents a new static logic implication algorithm. An improved implication procedure that fully takes advantage of the special context of static implication, the iterative method, and set algebra is described. The algorithm discovers at low cost many indirect implications which are not discovered by dynamic learning without tremendous time cost. The experimental results show that a very large number of indirect implications are found by our algorithm. The static implication procedure has many useful applications, one of which is static redundancy identification. Use of the static implications obtained from the algorithm in static redundancy identification for ISCAS85 combinational circuits resulted in a larger number of redundant faults identified than in previous methods.

I Introduction

Static logic implication, also called *static learning*[1], is a procedure which performs implications on both value assignments (0 and 1) for all nodes of a circuit. It is often included in the preprocessing phase of test generation and other applications [1]-[6]. For example, it is used in ATPG to avoid repetitive computation of signal assignments and accelerate the test pattern generation. Since the usual direct implications made by forward and backward propagation can be quickly determined during the dynamic learning phase, the emphasis of static learning should be put on indirect implications, those necessary assignments that cannot be found by simple forward and backward signal propagation [2]. Indirect implications play a critical role in many processes, such as multi-level logic optimization [4], redundancy identification [7][8], ATPG [2], and logic verification. A vast majority of indirect implications, especially *unilateral indirect implications*[2], can be easily found in static learning using the contrapositive law, while it is difficult, and sometimes practically impossible, to discover them in dynamic learning. Some previous work [9] in ATPG also showed that with a complete and efficient preprocessing phase, dynamic calculation of the logical dependencies among nodes is not required to process the vast majority of faults. Therefore, static learning is a very important preprocessing step.

A number of papers have dealt with implication procedures [1]-[4][9]-[12]. The learning procedures described in [3] and [5] can discover some indirect implications, but they are not sufficient for identifying large numbers of indirect implications. Rajski and Cox used a 16-value logic algebra and reduction list method to determine necessary assignments [9]. Chakradhar and Agrawal proposed a novel transitive closure based algorithm, which guarantees the identification of all implications of a partial set of node values [10][11]. The advantage of the algorithms proposed in [9]-[11] is that they not only identify necessary value assignments but also keep updating the logic dependencies between nodes, thus further speeding up computation of implications. However, the NP-hard nature of the problem of finding all the implications of setting a node to a particular value restricts the practicality of such complete algorithms. For the transitive closure algorithm [10][11], static learning of all the implications for all value assignments in the circuit needs repetitive computation of transitive closure, transforming boolean relationships among nodes, which is prohibitive in time and space. Another complete algorithm called recursive learning was proposed by Kunz and Pradhan [12]. In practical implementation of recursive learning, the depth of recursion must be restricted to keep the execution time within reasonable bounds. As a result, some implications may not be found.

In this work, we present an algorithm for static learning which discovers a very large number of indirect implications in reasonable time. Since few explicit results on static learning have been published (the result reported in SOCRATES [3] is too limited to be compared with the huge number of indirect implications found by our algorithm), we applied our static learning results to static redundant fault identification (RID) to show the efficacy of our algorithm. The RID procedure is based on the FIRE algorithm [13][14]. By applying our static implication results to RID, we obtained better results than those reported in [13][14].

*This research was supported in part by the Semiconductor Research Corporation under contract SRC 96-DP-109, in part by DARPA under contract DABT63-95-C-0069, and by Hewlett-Packard under an equipment grant.

The rest of the paper is organized as follows. Section II describes the basic concept behind our algorithm. Section III presents our algorithm. Section IV describes how the results of our algorithm can be used in redundant fault identification. Section V gives the exprimental results, and Section VI concludes the paper.

II Basic Concept

The proposed static implication algorithm was inspired by the single pass deductive fault simulation algorithm of Armstrong [15]. In deductive fault simulation, sets of faults are propagated from inputs of a gate to its output using set operations of intersection, union, and difference. In an analogous manner, we propagate sets of implications from inputs of a gate to its output using the familiar set operations. The basic concept is illustrated in Figure 1 with the use of a 2-input AND gate. In the figure, A_0 is the set of implications computed so far for setting node a to 0; A_1 is the set of implications for setting node a to 1. Similarly B_0 and B_1 are sets of implications for $b = 0$ and $b = 1$, respectively. The implication sets are "propagated" to the ouput c by set operations indicated in the figure. Clearly, C_0, the set of implications of setting c to 0, is the common node values implied by $a = 0$ and $b = 0$, hence the intersection. Similarly, implications of $c = 1$ are all node values implied by $a = 1$ and $b = 1$, hence the union. Similar rules can be derived for OR and NOT gates.

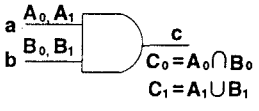

Figure 1: Implication example.

The single pass deductive fault simulation algorithm starts from the primary inputs and traverses the circuit in a levelized order, computing the sets of implications at each gate output. Similarly, the proposed implication algorithm starts from the primary inputs and processes the gates in a levelized order. In a single pass, this simple algorithm computes many implications. This algorithm can be enhanced by computing implications using other methods. For example, sets A_0 and A_1 in Figure 1 can be augmented by use of the contrapositive law, transitivity, and forward implication. After one pass of this algorithm, some sets may have changed, with new implications added. Therefore, another pass of the same algorithm may create even more implications. The final algorithm is based on these concepts and is presented in the next section. The key attributes of this algorithm are:

(a) It is iterative, unlike the recursive method in [11].
(b) Set operations are used to compute new implications.
(c) Nodes are processed concurrently for both 0 and 1 values.
(d) Transitive and contrapositive laws are integrated with set operations.

III Static Implication Algorithm

We first define a few terms and introduce some basic laws that will be used in our algorithm. We then present the algorithm and discuss implementation issues.

A Terms and basic laws

Direct and indirect implication are defined in [2]. We use the following terms to represent assignments and implications:

1. $[N, v]$: assign logic value v to node N;
2. $[M, w] \rightarrow [N, v]$: $[M, w]$ implies $[N, v]$;
3. $impl[N, v]$: set of implications resulting from setting node N to value v.

The following laws are used in the process of static implication:

1. Forward implication: If all the input values of a gate are known or one of the inputs is at the controlling value of the gate, then the output value of this gate can be uniquely determined from its input values. For example, for an AND gate, if one of the inputs is set to 0, then the output is 0; if all of the inputs are set to 1, then the output is 1.

2. Backward implication: Suppose we are generating implications of $[N, a]$. Let G be an unjustified gate with m unspecified input nodes S_i and a specified output node Y.

   if G is an AND gate:
   if $[Y, 0] \in impl[N, a]$,
   $impl[N, a] = impl[N, a] \cup (\bigcap_{i=1}^{m} impl[S_i, 0])$
   if $[Y, 1] \in impl[N, a]$,
   $impl[N, a] = impl[N, a] \cup (\bigcup_{i=1}^{m} impl[S_i, 1])$

   If $Y = 1$, then all gate inputs are 1, and we can add the implications of setting these inputs to 1 to our list of implications. If $Y = 0$, we find implications resulting from setting each input to 0, and since at least one input must be 0, we add the common implications found.

   if G is an OR gate:
   if $[Y, 1] \in impl[N, a]$,
   $impl[N, a] = impl[N, a] \cup (\bigcap_{i=1}^{m} impl[S_i, 1])$
   if $[Y, 0] \in impl[N, a]$,
   $impl[N, a] = impl[N, a] \cup (\bigcup_{i=1}^{m} impl[S_i, 0])$

3. Extended backward implication: For gate G with m unspecified input nodes S_i and a specified output node Y,

   if G is an AND gate:
   if $[Y, 0] \in impl[N, a]$ and $[Y, 0]$ is unjustified by gate inputs S_i, then
   $impl[N, a] = impl[N, a] \cup (\bigcap_{i=1}^{m} Forward\_Imply(impl[N, a] \cup impl[S_i, 0]))$

   $Forward\_Imply$ is a procedure performing forward implications on a set of node assignments.

if G is an OR gate:
if $[Y,1] \in impl[N,a]$ and $[Y,1]$ is unjustified by gate inputs S_i, then
$impl[N,a] = impl[N,a] \cup (\bigcap_{i=1}^{m} Forward\_Imply(impl[N,a] \cup impl[S_i,1]))$

4. Transitive law: If $[M,w] \rightarrow [N,v]$ AND $[N,v] \rightarrow [L,y]$, then $[M,w] \rightarrow [L,y]$. In set notation, if $[N,v] \in impl[M,w]$ and $[L,y] \in impl[N,v]$, then $[L,y] \in impl[M,w]$.

5. Contrapositive law: If $[M,w] \rightarrow [N,v]$, then $[N,\overline{v}] \rightarrow [M,\overline{w}]$. In set notation, if $[N,v] \in impl[M,w]$, then $[M,\overline{w}] \in impl[N,\overline{v}]$. This law enables the algorithm to discover unilateral indirect implications [2].

6. Conflicting assignments: If $[M,w] \rightarrow [N,v]$ AND $[M,w] \rightarrow [N,\overline{v}]$, then $[M,w]$ is an impossible setting. In other words, M will permanently hold the value $\overline{w}$. This law enables the algorithm to detect those nodes with constant values. Our algorithm includes conflict checking. If conflicts are not checked, the false values will create many useless new implications during execution of the algorithm, thus affecting the performance.

Extended backward implication discovers some indirect implications that cannot be discovered by simply applying the transitive and contrapositive laws. Although such implications usually occupy only a small part of the total implications found, they are $hard-to-find$ implications and play a critical role in speeding up an ATPG process. Usually, much time is wasted in a non-solution area of a decision tree due to some undiscovered global implications. Therefore, finding such indirect implications is one of the most important tasks of static learning.

We don't perform backward implication when $Y = 1$ and G is an AND gate, or when $Y = 0$ and G is an OR gate, since such backward implications can be found using the contrapositive law during the forward implication process. For example, in Figure 1, implications $[c,1] \rightarrow [a,1]$ and $[c,1] \rightarrow [b,1]$ can be found by the contrapositive law when performing forward implication on $(a,0)$ and $(b,0)$.

B The Algorithm

The basic laws decribed above form the core of our algorithm. The main function $SIMP$ and the two subroutines $Imply$ and $AddNew$ are shown in Figures 2, 3 and 4, respectively. Procedure $Imply$ creates new implications using the transitive law, simple forward implication, and extended backward implication. Procedure $AddNew$ checks if each new implication causes a conflicting assignment to a node. It also creates the contrapositive implication of the newly found implication.

Due to the special context of static learning, i.e., performing the learning procedure iteratively on all nodes using the contrapositive law, our algorithm discovers many indirect implications that are almost impossible to discover in dynamic learning. For example, in Figure 5, $impl[c,0] = \{[c,0],[f,0],[g,0],[m,0],[o,0],[s,0]\}$. Also, i is identified as a node with constant value 0. These facts are learned during the first two iterative phases. The following steps show how the implications in $impl[c,0]$ and the constant $[i,0]$ are discovered.

```
SIMP( )
    For every circuit node N
    [   impl[N,0] = {[N,0]};
        impl[N,1] = {[N,1]};
    While implications found
    [   For every circuit node N in levelized order
        [   Imply(N,0);
            Imply(N,1);
```

Figure 2: Main function.

```
Imply (N: node, v: logic-value)
    If [N,v] is marked impossible
       Then return;
    For every [M,w] in impl[N,v]
    [   impl[N,v] = impl[N,v] U impl[M,w];
                                /* by transitive law */
        AddNew( );
    Set circuit to values implied by values in impl[N,v];
    Forward_Imply( );
    AddNew( );
    For every [M,w] in impl[N,v] not implied by
        its gate input values
    [   If M is output of AND-gate and w = 0 or
           M is output of OR-gate and w = 1
        [   Then Extended_backward_imply(M,w);
                 AddNew( );
```

Figure 3: Imply.

```
AddNew( )
    For every new implication [X,a] found
    [   impl[N,v] = impl[N,v] U {[X,a]};
        impl[X,ā] = impl[X,ā] U [N,v̄]
                    /* by contrapositive law */
        If [X,ā] also belongs to impl[N,v]
        [   Then mark [N,v] as impossible;
                 return;
```

Figure 4: AddNew.

i. During the implication generation for $[c,0]$, $[c,0] \rightarrow [f,0]$ and $[c,0] \rightarrow [g,0]$ are learned by forward implication.

ii. During the implication generation for $[i,1]$, $[i,1]$ is detected as an impossible value assignment; therefore i is at constant value 0.

iii. During the implication generation for $[m,1]$, $[m,1] \rightarrow [c,1]$ is learned through forward and backward implication. Hence $[c,0] \rightarrow [m,0]$ is learned by the contrapositive law.

iv. During the implication generation for $[o,1]$, $[o,1] \rightarrow [c,1]$ is learned through forward and backward implication. Hence $[c,0] \rightarrow [o,0]$ is learned by the contrapositive law.

v. During the second iterative phase, in the implication generation for $[c,0]$, $[c,0] \rightarrow [s,0]$ is learned by set algebra performed at gate o, since $[s,0]$ is in $Forward\_Imply(impl[l,1] \cup impl[c,0]) \cap Forward\_Imply(impl[n,1] \cup impl[c,0])$.

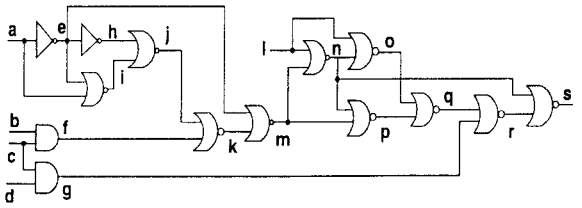

Figure 5: Example from c6288 benchmark circuit.

C Implementation Issues

Good data structures and a quick method for storing circuit information are indispensible in an efficient implementation of the implication procedure. The technique we use for clearing and storing circuit information is similar to that of PROOFS [16]. We assign each node in the circuit a tag and maintain a current tag value. When clearing the circuit, we simply increment the current tag value and update the tags on constant-valued nodes. Each time we assign a value to a node, we set its tag to the current tag value. In this way, we can tell whether the value of a node is currently valid. For set algebra operations, a fast mapping technique is used in our implementation. For example, when performing a set intersection operation for several sets of assignments, we map each set to the corresponding node assignments in the circuit. If a node is assigned the same value by each set, the corresponding node assignment is in the intersection. We also use an event-driven logic simulator to perform direct forward implication.

IV Redundant Fault Identification Using Static Implication

One successful application of static implication is static redundancy identification. Generally, there are three kinds of redundant faults: unexcitable faults, unpropagatable faults, and faults that cannot be excited and propagated simultaneously. A novel method used in FIRE [13][14] to identify redundancies without ATPG is to process conflicting values on the same line. A fault is redundant if it requires conflicting values on the same line as a necessary condition to be detected [13][14]. Based on the same principle, we applied our results of static implication to redundancy identification. We first introduce three definitions used in our static implication based redundancy identification procedure, *Simprid*:

1. Preimage: Let $[N,v]$ be an assignment. The preimage of $[N,v]$ is the set of all assignments that imply $[N,v]$.
   $preimage[N,v]$={$[M,w]$ | $[M,w] \rightarrow [N,v]$, M and N are nodes in a circuit, $w \in \{0,1\}, v \in \{0,1\}$}

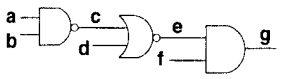

Figure 6: Preimage identification.

In our static implication algorithm, the contrapositive law is applied whenever a new implication is found. Therefore, the preimage of an assignment can be easily identified from the implication set of its opposite assignment. For example, consider the circuit in Figure 6, where $impl[a,0] = \{[a,0],[c,1],[e,0],[g,0]\}$. The preimage of $[a,1]$ is obtained by inverting all assignments in $impl[a,0]$: $preimage[a,1] = \{[a,1],[c,0],[e,1],[g,1]\}$. Therefore, faults $a/0, c/1, e/0$, and $g/0$ require assignment $[a,1]$ as a necessary condition for excitation.

2. Backcone: Let S be a set of nodes in the circuit. We can merge the nodes in S into a single node in the circuit graph. The backcone of S is a set of nodes dominated by the merged node corresponding to S in the collapsed graph.
   $backcone[S]$={P | every path from node P to a primary output passes through at least one node in S.}

3. Side-input: The side-inputs of a circuit node N are nodes that share the same successor node with N.

Let the side-inputs of all nodes that have noncontrolling values in $preimage[N,v]$ form a set S. All faults on the lines inside $backcone[S]$ will require $[N,v]$ as a necessary condition for propagation. For example, in Figure 6, a, c, and e have noncontrolling values in $preimage[a,1]$; b, d, and f are the side-inputs of a, c, and e, respectively. All the faults on the lines inside $backcone[\{b,d,f\}]$ require $[a,1]$ for propagation.

Thus, we can determine set_0 and set_1 for each node N in the circuit, where set_0 is the set of faults that require $[N,0]$ as a necessary condition for excitation or propagation, and similarly for set_1. The *Simprid* procedure is outlined in Figure 7.

```
Simprid( )
  redundantFaults = empty;
  For each node N in circuit
    set_0 = find faults that require [N,0] as a necessary
            condition for excitation (using preimage[N,0])
            and find faults that require [N,0] as a necessary
            condition for propagation (using backcone[S],
            where S is derived using preimage[N,0]);
    set_1 = find faults that require [N,1] for excitation or
            propagation in a similar manner;
    If [N,0] is impossible
      Then redundantFaults = redundantFaults U set_0 ;
    else if [N,1] is impossible
      Then redundantFaults = redundantFaults U set_1 ;
    else
      redundantFaults = redundantFaults U (set_0 ∩ set_1);
```

Figure 7: Static implication based RID procedure.

v Experimental Results

Both the proposed implication algorithm and the redundancy identification procedure were implemented in C on an HP 9000 J200 workstation with 256MB RAM. This section presents experimental results for ISCAS85 combinational benchmark circuits. Table 1 shows the results of our static learning algorithm. For each circuit, the total number of implications, the number of direct and indirect implications, the number of constant value assignments (*#Cons*), and the CPU time are shown. Constant value assignments are not counted as implications here. To find out the number of indirect implications obtained by our procedure, we ran a program that performs only direct implication on each node and subtracted the number of direct implications from the total number of implications found. We do not discriminate between stems and fanout branches. A stem and its fanout branches are considered to be the same node. A very large number of indirect implications are found by our algorithm, which reveals the attractive and promising prospect of this algorithm, especially in its application to ATPG for large sequential circuits. The performance of the algorithm may be further improved by removing some redundant computations in the iterative phases.

As reported in [17], by performing logic simulation of an exhaustive set of input vectors for c432, the upper limit of the number of static implications for c432 is found to be 2830. Our algorithm found 2806 implications.

Table 2 shows the distribution of implications. It is surprising to see that a single node assignment can imply hundreds of other nodes assignments.

To save the run time spent in extended backward implication, we also implemented the simplified backward implication (Rule 2 in Section III). Table 3 shows the results of this experiment. The run time is reduced at the expense of finding fewer implications. However, we still can find a large number of implications.

Table 1: Static learning results

Ckt	Number of Implications			# Cons	Time (sec)
	Total	Direct	Indirect		
c17	70	60	10	0	0.1
c432	2806	1608	1198	0	0.6
c499	7366	3838	3528	0	1.4
c880	7006	4881	2125	0	0.6
c1355	31990	18406	13584	0	3.6
c1908	47440	11046	36394	0	6.11
c2670	61658	17972	43686	11	15.5
c3540	313470	33758	279712	1	92.7
c5315	108130	36668	71462	1	51.4
c6288	30996	16507	14489	17	49.1
c7552	302064	64567	237497	4	182.2

Table 2: Distribution of implications

Ckt	Number of Nodes	Number of Assignments with n Implications			
		$n < 10$	$10 \leq n < 50$	$50 \leq n < 100$	$n \geq 100$
c17	13	26	0	0	0
c432	203	286	120	0	0
c499	275	402	100	48	0
c880	469	661	275	2	0
c1355	619	769	281	76	112
c1908	938	702	943	115	116
c2670	1566	1655	1120	247	99
c3540	1741	1139	1252	207	883
c5315	2608	2284	2532	282	117
c6288	2480	3636	1275	29	3
c7552	3827	2945	3421	511	767

Table 4 compares the results of our redundancy identification procedure *Simprid* and those of the original FIRE implementation [14]. The number of redundancies identified by each procedure is shown in the table for each circuit. For most of the ISCAS85 circuits where redundancies are found, the *Simprid* procedure identified more redundancies than FIRE did. The key to our success lies in the extremely large number of implications found in the static learning preprocessing phase.

vi Conclusion

This paper has presented a new static learning algorithm for use in static redundancy identification and other applications. Using the iterative method and the transitive law, our algorithm accomplishes transitive closure on the implications found during the procedure. Circuit information storage and set algebra are the main bottlenecks of our static learning algorithm. Efficient data structures were used to reduce the execution time of such operations. Experimental results show that most of the run time is spent in discovering a few implications with very

Table 3: Results of static learning with simplified backward implication

Ckt	Number of Implications			# Cons	Time (sec)
	Total	Direct	Indirect		
c17	70	60	10	0	0.1
c432	2734	1608	1126	0	0.4
c499	7366	3838	3528	0	1.4
c880	6990	4881	2109	0	0.5
c1355	31990	18406	13584	0	2.9
c1908	47290	11046	36244	0	4.8
c2670	61646	17972	43674	11	12.3
c3540	313192	33758	279434	1	77.5
c5315	107046	36668	70378	1	25.0
c6288	30024	16507	13517	17	11.0
c7552	301608	64567	237041	4	131.3

Table 4: Comparison of original FIRE with *Simprid*

Ckt	FIRE [14]		Simprid	
	# Red	Time (sec)	# Red	Time (sec)
c1908	6	1.8	4	2.2
c2670	29	1.5	39	2.5
c3540	93	11.9	105	14.7
c5315	20	2.8	20	26.6
c6288	33	1.3	34	2.7
c7552	30	4.7	42	15.6

long distance. If time is an important consideration, we may restrict the number of iterative steps or simplify the backward implication procedure in several ways. By doing so, we can reduce the run time and still find a very large number of implications. Improvements may also be achieved by avoiding redundant computations in iterative phases. If enough memory space for saving the circuit information is provided, the static learning procedure may be further enhanced by combining the iterative and the recursive methods [12] together.

The large number of implications found during the static implication preprocessing phase is the key to the superior performance of *Simprid* over FIRE. The *Simprid* procedure may be further improved by applying Theorems 1 and 2 in [7]. The experimental results for both static learning and redundancy identification only show the feasibility of our algorithm. There are many other applications for static learning, such as ATPG, tristate bus resolution, redundancy identification in sequential circuits, logic optimization, and illegal state identification in sequential circuits. Since static learning does not rely on any other procedure, it can be easily interfaced with many applications.

References

[1] M. H. Schulz and E. Auth, "Improved Deterministic Test Pattern Generation with Applications to Redundancy Identification," *IEEE Trans. Computer-Aided Design*, pp. 811-816, July 1989.

[2] W. Kunz and D. K. Pradhan, "Accelerated Dynamic Learning for Test Pattern Generation in Combinational Circuits," *IEEE Trans. Computer-Aided Design*, pp. 684-694, May 1993.

[3] M. H. Schulz, E. Trischler, and T. M. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," *IEEE Trans. Computer-Aided Design*, pp. 126-136, January 1988.

[4] W. Kunz and P. Menon, "Multi-Level Logic Optimization by Implication Analysis," *Proc. IEEE Int. Conf. Computer-Aided Design*, pp. 6-13, 1994.

[5] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Trans. Computers*, pp. 1137-1144, December 1983.

[6] P. Wohl and J. Waicukauski, "Test Generation for Ultra-Large Circuits Using ATPG Constraints And Test-Pattern Templates," *Proc. Int. Test Conf.*, pp. 13-20, October 1996.

[7] P. R. Menon and M. Harihara, "Redundancy Identification and Removal in Combinational Circuits," *Proc. Int. Conf. Computer Design*, pp. 290-293, October 1989.

[8] P. R. Menon and H. Ahuja "Redundancy Removal and Simplification of Combinatinal Circuits," *Proc. IEEE VLSI Test Symp.*, pp. 268-273, April 1992.

[9] J. Rajski and H. Cox, "A Method to Calculate Necessary Assignments in Algorithmic Test Pattern Generation," *Proc. IEEE Int. Test Conf.*, pp. 25-34, September 1990.

[10] S. T. Chakradhar and V. D. Agrawal, "A Transitive Closure Based Algorithm for Test Generation," *Proc. ACM/IEEE Design Automation Conf.*, pp. 353-358, June 1991.

[11] S. T. Chakradhar, V. D. Agrawal, and S. G. Rothweiler, "A Transitive Closure Algorithm for Test Generation," *IEEE Trans. Computer-Aided Design*, pp. 1015-1028, July 1993.

[12] W. Kunz and D. Pradhan, "Recursive Learning: An Attractive Alternative to the Decision Tree for Test Generation in Digital Circuits," *Proc. Int. Test Conf.*, pp. 816-825, September 1992.

[13] M. A. Iyer and M. Abramovici, "Low Cost Redundancy Identification for Combinational Circuits," *Proc. Int. Conf. VLSI Design*, pp. 315-318, January 1994.

[14] M. A. Iyer and M. Abramovici, "FIRE: A Fault-Independent Combinational Redundancy Identification Algorithm," *IEEE Trans. VLSI Systems*, pp. 295-301, June 1996.

[15] D.B.Armstrong, "A Deductive Method for Simulating Faults in Logic Circuits," *IEEE Trans. Computers*, pp. 464-471, May 1972.

[16] T. M. Niermann, W. T. Cheng, and J. H. Patel, "PROOFS: A Fast, Memory-Efficient Sequential Circuit Fault Simulator," *IEEE Trans. Computer-Aided Design*, pp. 198-207, February 1992.

[17] J. A. Newquist, "Fast Logic Implication Discovery," *M.S. Thesis*, University of Illinois at Urbana-Champaign, 1997.