Using Contrapositive Law in an Implication Graph to Identify Logic Redundancies

Kunal K. Dave
ATI Research INC.
Yardley, PA 19067, USA
kdave@ati.com

Vishwani D. Agrawal
Auburn University, Dept. of ECE
Auburn, AL 36849, USA
vagrawal@eng.auburn.edu

Michael L. Bushnell
Rutgers University, Dept. of ECE
Piscataway, NJ 08854, USA
bushnell@caip.rutgers.edu

Abstract – *Implication graphs are used to solve the test generation, redundancy identification, synthesis, and verification problems of digital circuits. We propose a new "oring" node structure to represent partial implications in a graph. The oring node is the contrapositive of the previously used "anding" node. An n-input gate requires one oring and one anding nodes to represent all partial implications. This implication graph is shown to be more complete and more compact compared to the previously published (n+1) anding node graph. Introduction of the new oring node finds more redundancies using the transitive closure method. The second contribution of the present work is a set of new algorithms to update transitive closure for every newly added edge in the implication graph associated with anding and oring nodes. For the ISCAS'85 benchmark circuit c1908, the new graph identifies 5 out of a total of 7 redundant faults. The best known previous implication graph procedure could only identify 2 redundant faults. We analyze the unidentified redundant faults and suggest a possible improvement.*

1. Introduction

An *implication graph* (IG) is a representation of a digital circuit in the form of a set of binary and higher-order relations between signals. It is used to solve systems of Boolean equations for test generation, redundancy identification, synthesis, and verification problems involving digital circuits. We focus on the application of implication graph to fault-independent redundancy identification.

Redundancy identification is useful in VLSI testing and design synthesis. There are two basic methods for identifying redundant faults: fault-dependent techniques and fault-independent techniques. *Fault-dependent techniques* are mainly ATPG based methods [4, 8, 12, 15, 21, 22, 31, 32, 33, 36], which target a particular fault at a time. Larrabee [23, 24], starting with the Boolean difference and Chakradhar *et al.* [5, 7, 9], with the neural network model, arrived at the satisfiability formulation of the ATPG problem. Both solved the problem with the help of *implication graphs. Fault-independent techniques* analyze the circuit topology and function without targeting a specific fault. To limit the complexity of the analysis, approximations and restrictions are often used. These methods can find some redundancies very quickly [20] but are not exhaustive in terms of finding all redundant faults. Such methods can be further classified as testability analysis [2, 16, 17, 30, 35] or implication based techniques [1, 10, 11, 14, 20, 28, 29].

Agrawal *et al.* proposed a fault-independent redundancy identification technique using an implication graph and transitive closure that analyzes circuit topology and function without targeting a specific fault [1]. In their work, they define observability variables, O_x, for every circuit line x. This variable assumes the logic value 1 only when x is observable at a primary output. Gaur *et al.* [14] presented a transitive closure algorithm for implication graphs that contain partial implications, where a vertex can assume the true state when all vertices that partially imply it become true. They represent these partial implication with the help of *anding* nodes [18, 37]. While the *anding* nodes potentially improve the representation, they forbid a straightforward computation of TC. The method of Gaur *et al.* provides improved results with a linear time complexity. Further improvements with all direct and partial implications and for node fixations were presented by Mehta *et al.* [28]. An implication graph containing full and partial implications and *anding* nodes is an incomplete representation of the Boolean function of the circuit, which motivated our work to further improve this representation.

We propose a new type of partial node, called *oring node*, to incorporate more complete logic information in the implication graph. We also propose a set of new algorithms to update transitive closure every time a new implication edge is added into the graph that contains signal nodes, *anding* nodes and *oring* nodes. We apply the new implication graph and the update algorithms to fault-independent redundancy identification and demonstrate improved performance.

2. Prior work

An *implication graph* (IG) is a representation of a digital circuit in the form of a set of binary and higher-order relations between signals. This graph has a node for every literal. Thus, a Boolean variable x is represented by two nodes, x and $\overline{x}$. A node can be *true* or *false*. For $x = 1$, the x node assumes the *true* state. For $x = 0$, $\overline{x}$ becomes *true*. Let us take an example of a two-input AND gate (Figure 1). The expanded Boolean false function [6] for this AND gate can be written as:

$$\overline{a}c + \overline{b}c + ab\overline{c} = 0 \qquad (1)$$

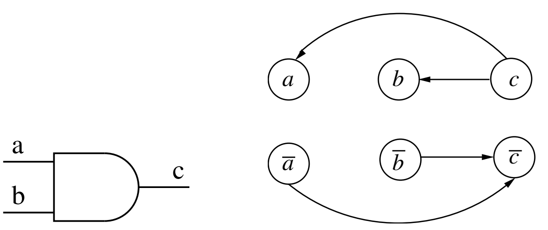

Figure 1: Direct implications for a 2-input AND gate.

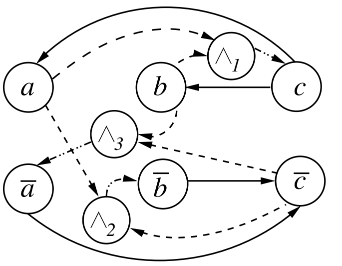

Figure 2: Partial implications for 2-input AND gate.

For Equation 1 to hold, all three terms on the left hand side must be 0. The first two terms show binary (pair-wise) relationships between signals. To make the first term $\overline{a}c = 0$, one of the following relations should be satisfied:

1. if $a = 0$ then $c = 0$
2. if $c = 1$ then $a = 1$

The first condition gives the implication, $\overline{a} \Rightarrow \overline{c}$ ($\overline{a}$ implies $\overline{c}$). We also obtain the implication $c \Rightarrow a$ from the second condition. Similarly, the second term gives the implications, $\overline{b} \Rightarrow \overline{c}$ and $c \Rightarrow b$. A two-variable "if ... then" clause is represented as a directed edge from a literal representing the "if" condition to another literal representing the "then" clause. The logical implications are expressed as *edges*. The binary relationships obtained from Equation 1 can be represented in the implication graph of *directed edges* as shown in Figure 1 [1, 3, 7].

An *enhanced implication graph* (EIG) was proposed by Henftling *et al.* [18, 37]. Gaur *et al.* [14] used it for redundancy identification. The term $ab\overline{c}$, a higher-order term in Equation 1, is a ternary relationship. To make the term $ab\overline{c} = 0$ one of the following relational conditions should be satisfied:

1. if $a = 1$ and $b = 1$, then $c = 1$
2. if $c = 0$ and $b = 1$, then $a = 0$
3. if $c = 0$ and $a = 1$, then $b = 0$

These relationships give *partial implications*. The implication graph for the AND gate with *anding* nodes is shown in Figure 2. The symbol $\wedge$ is used for the *anding* node in Figure 2. The traversal of the graph with *anding* nodes requires special consideration. We cannot traverse through an *anding* node unless we can arrive at it through all incoming edges. In general, for an n-input gate we require $n + 1$ *anding* nodes, each having n incoming edges and one outgoing edge. Also for each input signal of the gate an implicit observability AND gate expresses the observability related implications [1, 14]. For example, for signal a the observability is expressed as $O_a = bO_c$. Each of these n implicit gates requires $n + 1$ *anding* nodes with n incoming edges and one outgoing edge. Thus, to represent all the controllability and observability partial implications EIG requires $(n + 1)^2$ partial nodes.

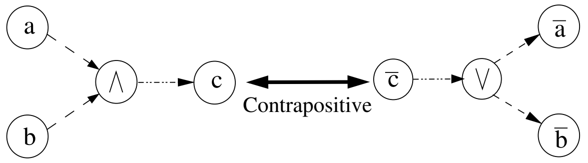

Figure 3: *Oring* node from an *anding* node.

3. Oring nodes

We introduce a new type of partial node called *oring* node. There exists a logical identity between two Boolean variables called the *contrapositive* law [34]:

$$(P \Rightarrow Q) \iff (\overline{Q} \Rightarrow \overline{P}) \qquad (2)$$

This means that if a variable P implies another variable Q then we conclude that the false state of Q implies the false state of P.

We introduce a new implication relation in the implication graph, the *OR implication*, that explicitly represents some contrapositive edges that are otherwise missing from implication graphs. We represent these relationships using an *oring* node, which is similar to the *anding* node used in several previous methods [13, 14, 26, 27, 28]. Let us again consider the example of a two input AND gate as shown in Figure 1 to derive the *oring* node. The expanded Boolean false function for this gate is given by Equation 1. As discussed in Section 2, we obtain two full implications from the first binary term on the left hand side of Equation 1 ($\overline{a} \Rightarrow \overline{c}$, $c \Rightarrow a$). If we analyze these two implications, we can conclude that implication $\overline{a} \Rightarrow \overline{c}$ is a contrapositive implication of $c \Rightarrow a$ according to Equation 2, and vice versa. Also, the same conclusion applies to the other two implications ($\overline{b} \Rightarrow \overline{c}$, $c \Rightarrow b$) obtained from the second term in the Boolean false function. We also have a ternary term in the Boolean false function, which produces partial implications as shown by an *anding* node on the left hand side in Figure 3. If we apply the contrapositive rule to the forward partial implication ($a \wedge b \Rightarrow c$) we can obtain the relation $\overline{c} \Rightarrow \overline{(a \wedge b)}$. According to *de Morgan's Law* [25] of Boolean algebra, we can establish the following relationships:

$$\overline{(P \vee Q)} \iff (\overline{Q} \wedge \overline{P}) \qquad (3)$$

and

$$\overline{(P \wedge Q)} \iff (\overline{Q} \vee \overline{P}) \qquad (4)$$

We use the *de Morgan's law* of Equation 4 to transform the term $\overline{(a \wedge b)}$ into $\overline{a} \vee \overline{b}$ and rewrite the forward partial implication $\overline{c} \Rightarrow \overline{(a \wedge b)}$ as $\overline{c} \Rightarrow \overline{a} \vee \overline{b}$.

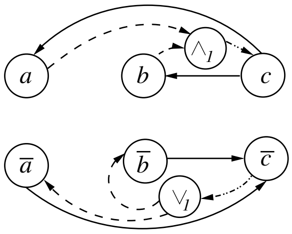

Figure 4: New implication graph of AND gate.

This means output $c = 0$ requires either input $a = 0$ or $b = 0$. This information can be represented in the implication graph as shown on the right hand side in Figure 3. Thus, for every *anding* node of the implication graph in Figure 2, we can find a corresponding contrapositive *oring* node.

Let us consider another *anding* implication ($a \wedge \overline{c} \rightarrow \overline{b}$) shown in Figure 2. The contrapositive *oring* node is ($b \Rightarrow \overline{a} \vee c$). This implication means that if signal $b = 1$ then either $a = 0$ or $c = 1$. It does not represent meaningful information in the implication graph as any value on signal b does not rely on signal a and also any logical conclusion cannot be drawn about the value on signal c just with signal $b = 1$ without any knowledge of the value on signal a. Similarly, no information can be inferred from contrapositive *oring* node, $a \rightarrow \overline{b} \vee c$, obtained from the *anding* node $a \wedge \overline{c} \rightarrow \overline{b}$. Thus, we ignore the implementations of these *oring* nodes in the implication graph. Also, we introduce an algorithm that uses *oring* nodes to obtain partial implications that were previously obtained by the other two *anding* nodes shown in Figure 2. The new implication graph (IG) for a two-input AND gate with all the controllability related partial implications is shown in Figure 4. In general, the proposed IG requires only one *anding* node and one *oring* node to represent controllability relations for a n-input logic gate. Also, one *anding* and one *oring* nodes are required to represent each observability AND gate as explained in Section 2.

Thus, the proposed implication graph requires $2(n + 1)$ partial nodes to represent all the controllability and observability related partial implications as compared $(n + 1)^2$ partial nodes used by EIG.

4. Transitive closure "Update" algorithms

In this section we first propose an algorithm that, given a transitive closure graph and a new full implication edge to be added, produces an updated transitive closure. This algorithm does not deal with partial implications. The algorithm is given below. Here G is an initial transitive closure to which an edge from source node v_s to destination node v_n is added. The routine $Update(G, v_s, v_n)$ returns G as the updated transitive closure.

Algorithm: **Update**

```
(1)   $Update(G, v_s, v_n)\{$
(2)       for each parent $P_i$ of source $v_s\{$
(3)           for each child $C_j$ of destination $v_n\{$
(4)               if (edge $P_i \rightarrow C_j$ does not exist)
(5)                   $addTcEdge(P_i, C_j);\}\}$
(6)   $\}/*Update*/$
```

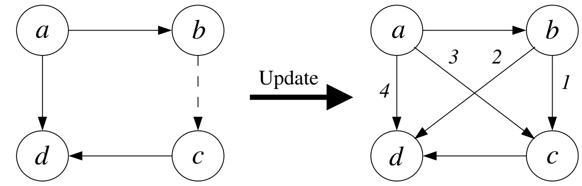

Figure 5: Transitive closure update for a new full edge using the **Update** algorithm.

Table 1: Parent and child node list for the graph of Figure 5 before adding the $b \rightarrow c$ edge.

Nodes	Parent List	Child List
a	a	$a,\ b,\ d$
b	$b,\ a$	b
c	c	$c,\ d$
d	$d,\ a,\ c$	d

where $addTcEdge(P_i, C_j)$ routine is only called when the condition of line (4) is *true*. It adds a transitive closure edge from node P_i to node C_j.

As an example, consider the graph shown on the left in Figure 5 without the dashed line edge. This graph is a transitive closure with four nodes and three edges (shown by solid lines). When a new edge from node b to node c (shown with a dashed arrow) is added, the $Update(G, b, c)$ routine is called with $v_s = b$ and $v_n = c$. This activates lines (1) to (6) in the algorithm shown above. The *for* loop in line (2) executes two times because node b has two parent nodes as shown in Table 1. The inner *for* loop in line (3) also executes two times as node c has two child nodes. In the first iteration, $P_i = b$ and $C_j = c$. There is no edge between nodes b and c, which satisfies the condition in line (4). Thus, the algorithm goes to line (5) and calls $addTcEdge\,(b, c)$ to add the transitive closure edge $b \Rightarrow c$. It follows similar steps for $b \rightarrow d$ and $a \rightarrow c$ edges in the second and third iterations and adds TC edges $b \Rightarrow d$ and $a \Rightarrow c$, respectively, as shown in Figure 5. For the edge $a \rightarrow d$, the condition in line (4) is not satisfied as there already exists an edge from node a to node d. Thus, the algorithm does not call $addTcEdge(a, d)$ for this edge. The updated transitive closure graph is shown on the right in Figure 5.

4.1. Update for partial implications

We propose transitive closure update algorithms that convert partial implications into possible full implications using *anding* and *oring* nodes. We explain these algorithms using an example. Figure 6 shows an example where two AND gates are fed by the same input signals a and b. In our technique, the computation starts with all binary signal nodes, *anding* nodes with only the partial incoming edges and one outgo-

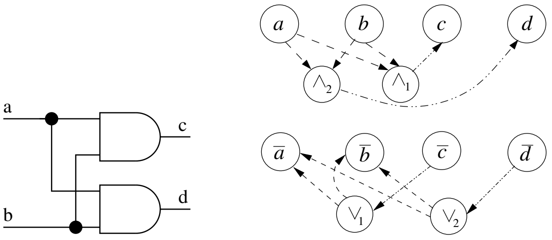

Figure 6: Implication graph of a circuit (left) before the application of *update* algorithms.

ing edge, and *oring* nodes with the partial outgoing edges and one incoming edge (see Figure 6). Now, every time a new transitive closure edge is added because of the **Update** routine, we check if the destination node v_n is a parent of any *anding* node $\wedge_x$. If this condition is satisfied we check the parent nodes of the *anding* node $\wedge_x$. For all of the nodes in this list of parent nodes, we try to find a common predecessor node G_p. If such a node is found we can put a transitive closure edge from the node G_p to the child node of the *anding* node $\wedge_x$ (*procedure 1*).

We propose another algorithm to traverse the *oring* nodes. Every time a new transitive closure edge is added by the **Update** routine, we check if the source node v_s is a child of any *oring* node $\vee_x$. If this condition is satisfied, we check the child nodes of the *oring* node $\vee_x$. For all of the nodes in this list of child nodes, we try to find a common successor node G_c. If such a node is found we put a transitive closure edge from the parent of the *oring* node to that common successor node G_c (*procedure 2*).

Figure 6 shows an implication graph of the circuit before *update* algorithms are executed. This graph is a transitive closure of itself. In this graph each node is considered as a parent and a child of itself as this is a basic property of any transitive closure. The first edge extracted from the logic network is from node c to node a, as we know that if $a = 0$ then $c = 0$. This invokes $Update(G, c, a)$. There is only one parent of node c and only one child of node a in this iteration. Thus, both the *for* loops in the **Update** routine are executed only once and this adds a TC edge $c \Rightarrow a$. The destination node $v_n = a$ is a parent node of an *anding* node $\wedge_1$ and $\wedge_2$. Thus, we check the parent node list of the *anding* node $\wedge_1$. We try to find a common predecessor node G_p, in this iteration no such node will be found as the predecessor list of the parent node a of node $\wedge_1$ contains $P_a = \{a, c\}$ and predecessor list of the parent node b of node $\wedge_1$ contains node $P_b = \{b\}$. Thus, no TC edge will be added. Similarly, no TC edge will be added because of the *anding* node $\wedge_2$. When TC edge $b \Rightarrow c$ is added using **Update**, we again try to find common grandparent of the *anding* node $\wedge_1$ as the new TC edge ends at a parent node of this *anding* node. In this iteration, $P_a = \{a, c\}$ and $P_b = \{b, c\}$. Intersection of these two lists gives a common predecessor node c. Thus, TC edge can be added from node c to d (thick edge). Similar steps are carried out for edges $d \rightarrow a$ and $d \rightarrow b$, which in turn adds TC edges $d \Rightarrow a$ and $d \Rightarrow b$ using **Update** routine and $d \Rightarrow c$ using *procedure 1* described above.

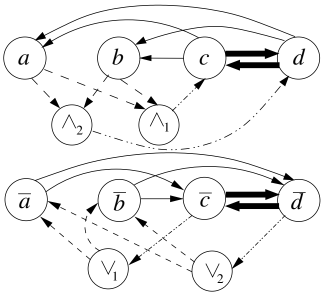

Figure 7: A portion of the transitive closure after **Update** algorithms are applied to the implication graph for the circuit given in Figure 6. Bold arrows show edges added by *procedures 1 and 2*.

When we extract an implication from node $\overline{a}$ to $\overline{c}$, new TC edge will be added $\overline{a} \Rightarrow \overline{c}$ because of the **Update** routine. The source node $\overline{a}$ of this TC edge is a child node of *oring* nodes $\vee_1$ and $\vee_2$. Thus *procedure 2* will be activated. In this iteration, no common grandchild will be found for any of the *oring* node. When the TC edge $\overline{b} \Rightarrow \overline{c}$ is added using **Update**, we again try to find common grandchild of the *oring* node $\vee_1$ as the new TC edge starts at a child node of the *oring* node. In this iteration, child list of node $\overline{a}$, $C_{\overline{a}} = \{\overline{a}, \overline{c}\}$ and that of node $\overline{b}$ is $C_{\overline{b}} = \{\overline{b}, \overline{c}\}$. The common node in these two lists can be found as node $\overline{c}$. Thus, a TC edge can be added from node $\overline{d} \Rightarrow \overline{c}$ according to *procedure 2* (thick edge). Similarly, a TC edge $\overline{c} \Rightarrow \overline{d}$ will be added as a result of new edges $\overline{a} \Rightarrow \overline{d}$ and $\overline{b} \Rightarrow \overline{d}$. The complete transitive closure after the application of **Update** algorithm, *procedure 1* and *procedure 2* is shown in Figure 7.

Thus, we can see that as compared to transitive closure edges $c \Rightarrow d$ and $d \Rightarrow c$ found by EIG [14], our IG and new algorithms find additional transitive closure edges $\overline{c} \Rightarrow \overline{d}$ and $\overline{d} \Rightarrow \overline{c}$. Since more implication edges can be found using the new implication graph and proposed update algorithms, we may identify more redundancies in the logic circuit.

4.2. Extended use of the *oring* nodes

In the previous subsection, we discussed the use of the *oring* node and the new implication graph to find extra redundancies. We also use the *oring* nodes to obtain backward partial implications that were previously obtained using *anding* nodes. Consider the example circuit shown in Figure 8. It can be seen that the first fanout branch of input signal a cannot be observed at the primary output e. Consider the transitive closure graph (obtained using previous methods [13, 26] and an additional *oring* node structure) shown in Figure 8. Here, observability node O_{ac} (observability of fanout branch of a feeding into gate c) implies nodes b and O_c. Also, observability of node O_c implies node $\overline{d}$. We can find the TC edge $O_{ac} \Rightarrow \overline{d}$ using any transitive closure calculation routine for the implication edge $O_{ac} \rightarrow O_c$. From the circuit topology we can obtain the *anding* node $\wedge_1$,

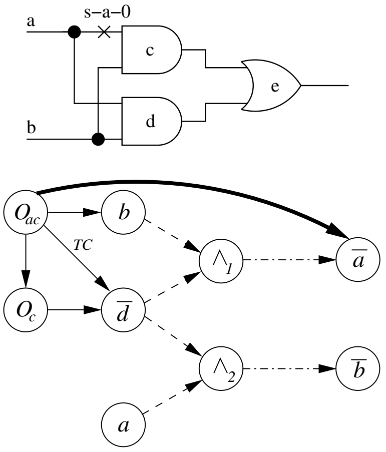

Figure 8: Transitive closure of an example circuit (top) using a previous method [28].

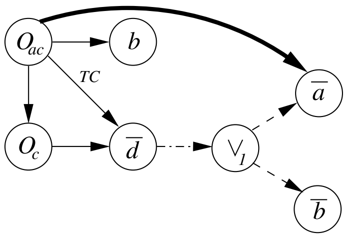

Figure 9: Transitive closure of IG for the circuit of Figure 8 using the proposed method.

where node b and node $\overline{d}$ together through this *anding* node imply node $\overline{a}$. Now nodes b and $\overline{d}$ feed into the *anding* node $\wedge_1$ through partial edges, which implies node d. Thus, we add the TC edge $O_{ac} \Rightarrow \overline{a}$. This gives us a condition to identify the *s-a-0* redundancy on line *ac*. We classify this type of *anding* node as a backward *anding* implication as it gives a partial implication from an output node ($\overline{d}$) to an input node ($\overline{a}$). We use the *oring* node structure to obtain all of the partial implications that were obtained using this type of backward *anding* nodes. Let us consider the transitive closure graph shown in Figure 9 without the *anding* nodes. Here we have removed the *anding* node $\wedge_1$ from the graph. Now, if we consider the same TC without the *anding* node we still have the TC edge $O_{ac} \Rightarrow \overline{d}$. As we can see, node $\overline{d}$ implies either node $\overline{a}$ node $\overline{b}$ through the *oring* node $\vee_1$. This indirectly says that observability of node O_{ac} implies either signal $a = 0$ or $b = 0$. Now, if we check the child nodes of O_{ac}, they have node b in them. That means the O_{ac} cannot imply node $\overline{b}$ through the *oring* node $\vee_1$. That in turn implies signal $a = 0$, which gives a transitive closure edge $O_{ac} \Rightarrow \overline{a}$. Here, we obtained the same TC edge that was obtained using the *anding* node in Figure 8. The information contained in the backward *anding* node $\wedge_2$ $((\overline{d} \wedge a) \rightarrow \overline{b})$ of Figure 8 is also included in the same *oring* node $\vee_1$. In general, from the three *anding* nodes (one forward and two backward) used to represent controllability partial relations in Figure 2, we only require one forward *anding* node $((a \wedge b) \rightarrow c)$ and one *oring* node in our new implication graph, which provide all of the necessary information as explained in the previous section. Similarly, backward *anding* nodes used to represent observability partial relations for each input signal can also be removed.

5. Results

Results for ISCAS'85 and ISCAS'89 benchmark circuits are shown in Table 2. Stand-alone combinational parts of ISCAS'89 sequential circuits were considered. The first column of the result table shows the name of the benchmark circuit for which results from various programs are compared with respect to the redundant faults identified and their respective CPU times. There are no aborted faults in TRAN for all circuits in Table 2. The next two columns show the redundant faults identified and the CPU times in seconds for the ATPG tool TRAN [8] by Chakradhar *et al.*, which uses transitive closure for test generation. The next two columns list the results of FIRE [19] by Iyer and Abramovici. The next two columns show the results of the transitive closure algorithm with some of the partial implications by Gaur *et al.* [14]. The next two columns show the results of the transitive closure algorithm with all of the partial implications obtained using *anding* nodes by Mehta *et al.* [28]. The last two columns are the results obtained using our new implication graph and the proposed algorithms.

We identified 5 out of the total of 7 redundant faults in the c1908 combinational benchmark circuit in 5.7 CPU seconds, while the ATPG tool TRAN [8] identifies all of the 7 redundant faults in 13.0 CPU seconds. FIRE [20] identifies 6 redundant faults in 1.8 CPU seconds, TC_G [13] identifies 2 redundant faults in 0.9 CPU seconds, and TC_M [28] identifies 2 redundant faults in 3.2 CPU seconds. We identify 65 redundant faults in the c7522 combinational benchmark circuit in 17.7 CPU seconds, while the ATPG tool TRAN [8] identifies all of the 131 redundant faults in 308.0 CPU seconds. FIRE [20] identifies 30 redundant faults in 4.7 CPU seconds, TC_G [13] identifies 34 redundant faults in 5.8 CPU seconds, and TC_M [28] identifies 51 redundant faults in 11.5 CPU seconds. We identify 51 redundant faults in the s1238 combinational benchmark circuit in 5.4 CPU seconds, while TRAN identifies all 69 redundant faults in 17.4 CPU seconds. FIRE identifies 6 redundant faults in 1.9 CPU seconds, TC_G identifies 6 redundant faults in 0.6 CPU seconds, and TC_M identifies 20 redundant faults in 2.6 CPU seconds. We identify more redundant faults then all other fault-independent techniques in all of the benchmark circuits, with a comparable CPU time of execution. In some benchmark circuits such as c5315, c6288, s349, s713, and s1423 we identify almost as many redundant faults as TRAN does, but we do it much faster.

Two unidentified redundant faults of c1908 are shown in Figure 10. Activation of these faults requires a logic 0 on signal 949 or on 953. In either case, signals 74, 887 and 886 become unobservable. Although each fanout branch of 979 is unobservable, lacking an observability relation between a stem and

Table 2: Combinationally redundant faults identified in ISCAS'85 and ISCAS'89 circuits.

Circuit	Total faults	Number of redundant faults identified and run time									
		TRAN [8]		*FIRE* [20]		TC_G [14]		TC_M [28]		Our method	
		Red. faults	CPU s Sparc 5	Red. faults	CPU s Sparc 2	Red. faults	CPU s Sparc 5	Red. faults	CPU s Sparc 5	Red. faults	CPU s Sparc 5
c432	524	4	0.8	-	-	0	0.2	0	0.2	2	0.3
c499	758	8	1.8	-	-	0	0.2	0	1.3	0	1.5
c880	942	0	4.0	-	-	0	0.4	0	0.4	0	0.5
c1355	1574	8	11.0	-	-	0	0.9	0	1.9	0	2.3
c1908	1879	7	13.0	6	1.8	2	0.9	2	3.2	5	5.7
c2670	2747	115	95.2	29	1.5	25	1.5	59	4.0	69	6.0
c3540	3428	131	24.9	93	11.9	74	6.2	110	16.2	111	17.0
c5315	5350	59	32.3	20	2.8	32	3.4	58	3.9	58	5.2
c6288	7744	34	38.0	33	1.3	31	1.8	34	7.2	34	10.3
c7552	7550	131	308.0	30	4.7	34	5.8	51	11.5	65	17.7
s349	350	2	0.3	2	0.2	2	0.2	2	0.2	2	0.2
s444	474	14	0.4	11	0.2	8	0.2	10	0.3	12	0.7
s713	581	38	3.1	32	0.1	35	0.3	38	0.6	38	0.9
s1238	1355	69	17.4	6	1.9	6	0.6	20	2.6	51	5.4
s1423	1515	14	8.5	5	0.3	8	0.7	12	1.0	13	2.3
s1494	1506	12	3.7	1	1.1	1	0.8	2	8.8	3	6.8
s5378	4603	40	73.0	34	3.7	22	3.0	23	8.3	26	8.4
s9234	6927	452	803.7	165	20.6	135	11.2	233	106.0	250	140.0
s13207	9815	151	806.5	55	23.2	60	13.6	77	158.8	78	190.0

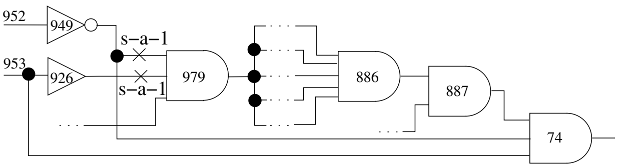

Figure 10: Two redundant faults of c1908 that were not identified by the implication garph algorithm. Some fanins are omitted for clarity.

its fanout branches, we are unable to deduce that 979 is unobservable. Kirkland and Mercer [21] state that observation of a signal requires the simultaneous observation of all of its absolute dominators. *Absolute dominators* of signal 979 are 886, 887 and 74. No path from 979 can reach any primary output without passing through all of its absolute dominators. If we enhance the implication graph by placing implication arcs from the *false observability nodes* ($\bar{O}$) of any or all absolute dominators to the $\bar{O}$ node of a corresponding dominated fanout stem signal, then the two redundant faults will be identified.

6. Conclusion

We derived a new partial implication structure called the *oring* node to represent logical dependencies of signals in the implication graph. Results indicate that for a fault-independent redundancy identification technique the proposed implication graph obtains better results. Also, as compared to the $(n+1)^2$ partial implication *anding* nodes used by Mehta *et al.* [26, 28], we only use $2(n+1)$ partial implication nodes (*anding* + *oring* nodes) for an n-input logic gate. Our new algorithms dynamically update a transitive closure every time an implication edge is added. These algorithms evaluate *anding* nodes and *oring* nodes to convert partial implications into full implications, which in turn add new transitive closure edges to the graph. We devise our algorithm such that it uses the proposed *oring* nodes to get all of the implications that were previously obtained using backward implication *anding* nodes. Repeated use of these algorithms constructs a transitive closure from an implication graph with full and partial implications. Once the transitive closure graph is calculated, we follow the same procedure used by previous techniques to find redundancies in the digital circuit.

Many unidentified redundant faults are on fanout stems. Their analysis suggests further improvements. However, redundancy identification has an exponential complexity. So, any implication graph or transitive closure based method with polynomial complexity will fail to identify some redundant faults. The present work provides improvements over previous algorithms in the polynomial complexity class.

References

[1] V. D. Agrawal, M. L. Bushnell, and Q. Lin, "Redundancy Identification Using Transitive Closure," in *Proc. of the 5th Asian Test Symp.*, November 1996, pp. 4–9.

[2] V. D. Agrawal and M. R. Mercer, "Testability Measures – What Do They Tell Us?," in *Proc. of the International Test Conf.*, November 1982, pp. 391–396.

[3] M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits.* Boston, MA: Kluwer Academic Publishers, 2000.

[4] M. L. Bushnell and J. Giraldi, "A Functional Decomposition Method for Redundancy Identification and Test Generation," *Journal of Electronic Testing:*

Theory and Applications, vol. 10, no. 3, pp. 175–195, June 1997.

[5] S. T. Chakradhar, *Neural Network Models and Optimization Methods for Digital Testing.* PhD thesis, Rutgers University, CS Dept., October 1990.

[6] S. T. Chakradhar, V. D. Agrawal, and M. L. Bushnell, "Neural Net and Boolean Satisfiability Models of Logic Circuits," *IEEE Design and Test of Computers*, vol. 7, no. 5, pp. 54–57, October 1990.

[7] S. T. Chakradhar, V. D. Agrawal, and M. L. Bushnell, *Neural Models and Algorithms for Digital Testing.* Boston, MA: Kluwer Academic Publishers, 1991.

[8] S. T. Chakradhar, V. D. Agrawal, and S. G. Rothweiler, "A Transitive Closure Algorithm for Test Generation," *IEEE Trans. on Computer-Aided Design*, vol. 12, no. 7, pp. 1015–1028, July 1993.

[9] S. T. Chakradhar, M. L. Bushnell, and V. D. Agrawal, "Automatic Test Generation Using Neural Networks," in *Proc. of the International Conf. on Computer-Aided Design*, November 1988, pp. 416–419.

[10] K. K. Dave, "Using Contrapositive Rule to Enhance the Implication Graphs of Logic Circuits," Master's thesis, Rutgers University, ECE Dept., May 2004.

[11] K. K. Dave, V. D. Agrawal, and M. L. Bushnell, "Using Contrapositives to Enhance the Implication Graph of Logic Circuits," in *Proc. of the 13th IEEE North Atalantic Test Workshop*, June 2004, pp. 56–63.

[12] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," in *Proc. of the International Fault-Tolerant Computing Symp.*, June 1983, pp. 98–105.

[13] V. Gaur, "A New Transitive Closure Algorithm to Identify Redundancies in Logic Circuits," Master's thesis, Rutgers University, ECE Dept., January 2002.

[14] V. Gaur, V. D. Agrawal, and M. L. Bushnell, "A New Transitive Closure Algorithm with Applications to Redundancy Identification," in *Proc. of the 1st International Workshop on Electronic, Design and Test Applications (DELTA'02)*, January 2002, pp. 496–500.

[15] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Trans. on Computers*, vol. C-30, no. 3, pp. 215–222, March 1981.

[16] L. H. Goldstein, "Controllbility/Observability Analysis of Digital Circuits," *IEEE Trans. on Circuits and Systems*, vol. CAS-26, no. 9, pp. 685–693, September 1979.

[17] J. Grason, "TMEAS, a Testability Measurement Program," in *Proc. of the 16th Design Automation Conf.*, 1979, pp. 156–161.

[18] M. Henftling and H. Wittmann, "A New Data Structure to Solve the Satisfiability Problem in Digital Circuits," *Archiv fiir Elecktronik Und Obertragungstechnik*, vol. 49, no. 1, pp. 29–43, January 1995.

[19] M. A. Iyer and M. Abramovici, "Low-cost Redundancy Identification for Combinational Circuits," in *in Proc. of 7th International Conf. on VLSI Design*, January 1994, pp. 315–318.

[20] M. A. Iyer and M. Abramovici, "FIRE: A Fault-Independent Combinational Redundancy Identification Algorithm," *IEEE Transactions on VLSI Systems*, vol. 4, no. 2, pp. 295–301, June 1996.

[21] T. Kirkland and M. R. Mercer, "A Topological Search Algorithm for ATPG," in *Proc. of the 24th Design Automation Conf.*, June-July 1987, pp. 502–508.

[22] W. Kunz and D. K. Pradhan, "Recursive Learning: An Attractive Alternative to the Decision Tree for Test Generation in Digital Circuits," in *Proc. of the IEEE International Test Conf.*, September 1992, pp. 816–825.

[23] T. Larrabee, "Efficient Generation of Test Patterns Using Boolean Difference," in *Proc. of the International Test Conf.*, August 1989, pp. 795–801.

[24] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *IEEE Transactions on Computer-Aided Design*, vol. 11, no. 1, pp. 4–15, January 1992.

[25] M. M. Mano, *Digital Logic and Computer Design.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1979.

[26] V. J. Mehta, "Redundancy Identification in Logic Circuits using Extended Implication Graph and Stem Unobservability Theorems," Master's thesis, Rutgers University, ECE Dept., Piscataway, New Jersey, May 2003.

[27] V. J. Mehta, V. D. Agrawal, and M. L. Bushnell, "Theorems on Redundancy Identification," in *Proc. of the 12th North Atlantic Test Workshop*, May 2003.

[28] V. J. Mehta, K. K. Dave, V. D. Agrawal, and M. L. Bushnell, "A Fault-Independent Transitive Closure Algorithm for Redundancy Identification," in *Proc. of the 16th International Conf. VLSI Design*, January 2003, pp. 149–154.

[29] P. R. Menon, H. Ahuja, and M. Harihara, "Redundancy Identification and Removal in Combinational Circuits," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 5, pp. 646–651, May 1994.

[30] I. M. Ratiu, A. Sangiovanni-Vincentelli, and D. O. Pederson, "VICTOR: A Fast VLSI Testability Analysis Program," in *Proc. of the IEEE International Test Conference*, November 1982, pp. 397–401.

[31] J. P. Roth, "Diagonosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development*, vol. 10, no. 4, pp. 278–291, July 1966.

[32] M. H. Schulz, E. Trischler, and T. M. Serfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," *IEEE Trans. on Computer-Aided Design*, vol. CAD-7, no. 1, pp. 126–137, January 1988.

[33] J. P. M. Silva and K. A. Sakallah, "Grasp – A New Search Algorithm for Satisfiability," in *Proc. of the International Conf. on Computer-Aided Design*, November 1996, pp. 220–227.

[34] D. F. Stanat and D. A. McAllister, *Discrete Mathematics in Computer Science.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1977.

[35] J. E. Stephenson and J. Grason, "A Testability Measure for Register Transfer Level Digital Circuits," in *Proc. of the Int. Symp. on Fault Tolerant Computing, Pittsburgh, PA*, June 1976, pp. 101–107.

[36] P. Tafertshofer, A. Ganz, and K. J. Antreich, "IGRAINE – An Implication GRaph bAsed engINE for Fast Implication, Justification and Propagation," *IEEE Trans. Computer-Aided Design*, vol. 19, no. 8, pp. 648–655, August 2000.

[37] P. Tafertshofer, A. Ganz, and M. Henftling, "A SAT-Based Implication Engine for Efficient ATPG, Equivalence Checking and Optimization of Netlists," in *Proc. of the International Conf. on Computer-Aided Design*, November 1997, pp. 648–655.