Efficient Implication - Based Untestable Bridge Fault Identifier*

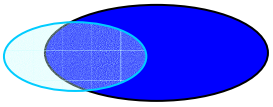
Manan Syal[†], Michael S. Hsiao[†], Kiran B. Doreswamy[††] and Sreejit Chakravarty[‡]
[†] Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA.
[††]Intel Corporation, Portland, Oregon.
[‡] Intel Architecture Group, Intel Corporation, Santa Clara, CA.

Abstract: *This paper presents a novel, low cost technique based on implications to identify untestable bridging faults in sequential circuits. Sequential symbolic simulation [1] is first performed, as a preprocessing step, to identify nets which are uncontrollable to a specific logic value. Then, an implication-based analysis is carried out for each fault to determine if a particular fault is testable or not. We also use information about the untestable stuck-at faults to filter out some bridges early in the analysis process. The application of our technique to ISCAS '89 sequential benchmark circuits and a few industrial circuits showed that a large number of untestable bridges could be identified at a low cost, both in terms of memory and execution time.*

1. Introduction: A bridge fault between two or more nets refers to a defect where the nets are unintentionally shorted together. While the size of the more commonly studied stuck-at fault list is of the order of the size of the circuit, bridging fault lists can potentially be of polynomial order in the size of the circuit. Furthermore, the cost of bridge-fault ATPG can exceed that of the stuck-at counterpart. Thus, it would be beneficial to quickly identify bridging faults that are redundant/untestable, so that tools such as bridging fault ATPG engines or bridging fault simulators do not waste time targeting these faults. Our work focuses on efficient identification of such untestable bridging faults.

Algorithms have been proposed earlier for two node [2] and multimode [3] bridge fault extraction/analysis. In our approach, we limit the analysis to two-node bridges. Also, we assume the bridge fault to be present between the outputs of two nodes, and no bridges internal to a Boolean gate are considered for analysis. Resistance of a bridge determines the kind of fault effect the bridge produces in a circuit. Bridges of high resistance type cause the outputs of the succeeding gates to have a delayed transition, leading to a delay fault, while bridges having low resistance (hard short) cause the static behavior of the circuit to change. In our analysis, we assume the bridge fault to be of the "hard short" type, which causes the functionality of the circuit to change. A lot of models have been proposed for the analysis of such kind of bridges. Stuck-at faults have been used to model the behavior of bridges [4], along with the voting [5] and the biased voting method [6], which are the more popular modeling methods. The model we use for our purpose of identifying untestable bridges is based on the wired and the dominant fault models [7].

Untestable bridges are faults for which no test pattern exists that can either excite the fault or propagate the fault effect to an observable point. Current automatic test pattern generators (ATPGs) waste a lot of time on such faults before identifying them as untestable or aborting on them. Although much work has been presented for the identification of untestable stuck-at faults [8][9][10], there hasn't been any significant contribution towards the identification of untestable bridges in sequential circuits. Thus, to the best of our knowledge, our work on the identification of untestable bridges using implications is the first of its kind. Our implementation is based on the representation of a sequential circuit as an iterative logic array (ILA) of fixed length [11]. For each bridge, we identify a set of conditions that are necessary for the fault to be testable. We imply these set of conditions within the ILA and use this information to determine whether the bridge can be declared as untestable or not. We evaluated our tool against the larger ISCAS '89 circuits along with a few industrial circuits. We validated and compared these results with our internal ATPG engine for bridging faults. Interestingly, it was observed that we could identify more untestable faults using this new tool than the ATPG tool could identify. The relationship between the untestable faults identified by the ATPG tool and our new implications based tool is shown in Figure 1.



Unt. faults identified only by ATPG
Unt. faults identified only by implications based tool
Unt. faults identified by both ATPG and implications based tool

Figure1: Relationship between the untestable faults identified by ATPG and the implications based tool

It can be seen that ATPG can identify some untestable bridges which the implication-based technique fails to capture (□); however, the implications based technique can identify a lot of untestable faults which the ATPG tool aborts on (■).

Rest of the paper is organized as given below:
Section 2 gives an introduction to logic implications, the symbolic simulation we use to identify uncontrollable nets and the fault model(s) we use to identify the behavior of a bridging fault. Section 3 describes our approach and algorithm, with section 4 reporting the results obtained for ISCAS '89 sequential circuits, and a few industrial circuits. Finally, section 5 concludes the paper.

2. Preliminaries

2.1 Logic Implications:

Static implications are the implications associated with binary logic values present at the output of every gate in the circuit. Static implications consist of direct, indirect and extended backward implications. Direct implications of a gate 'g' are the implications associated with the gates directly connected to 'g'. Direct implications are easy to compute, unlike Indirect and extended backward

* *This research was performed while the first two authors were at Intel during the period extending from 05/02 to 08/02. This work is also supported in part by NSF grant #0196470*

implications which require extensive use of transitive and contrapositive properties associated with implications [10].
The concept of direct and indirect implications can be understood from the following example:
Example 1: Consider the circuit shown in Figure 2.

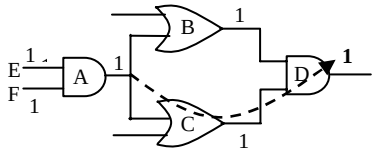


Figure 2: Illustration of direct and indirect implications

Consider the implications of gate A = 1 with respect to its directly connected nodes.

A = 1 → { (B = 1), (C = 1), (E = 1), (F = 1) }

This set, along with A = 1 forms the direct implications for node A = 1.
Now, nodes B = 1 and C = 1 do not individually imply any value on node D. However, together, they imply D = 1. Thus, A = 1 indirectly implies D = 1. Hence D = 1 becomes an indirect implication associated with gate A = 1.

Another notion associated with implications is that of backward and forward implications. Let's say we need to find the implications of gate $g = v$ (v = 0 or 1). All implications associated with the input cone of gate *g* form the backward implications for *g* and all implications associated with the output cone of *g* form the forward implications of *g*. This concept is shown with the aid of an AND gate in Figure 3.

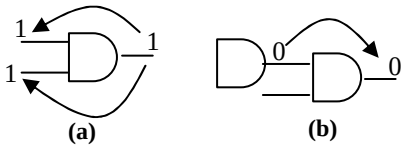


Figure 3: (a) backward implications (b) forward implications

Although the graphical representation of implications proposed in [10] is a memory efficient approach, it would still blow up for large (sequential) industrial circuits (with size > 100K gates). Also, the use of extended backward implications on such circuits would prove to be too expensive both in terms of memory and execution time.
Thus, in our application, we:

a) Do not pre-compute and store the implications associated with each gate in the circuit. We compute the implications of a gate on a need basis, i.e. only for gates that are involved in a bridge. We simply allocate the maximum space that can possibly be required to store the implications of a gate, or for the group of gates involved in a bridge, and we re-use that space for every fault we analyze.
b) Do not compute extended backward implications corresponding to every unjustified gate in the implication list of a gate [10], because it proves to be too expensive.

It may be argued that:

a) If a gate *g* is involved in a bridge with many other gates, then we might have to re-compute the implications of *g* each time we analyze a bridge in which *g* is involved. Since we don't store the implications of all gates upfront, this duplication of work may cost us in terms of time. However, since we compute the implications of a gate only on a need-basis, i.e. only if the gate is involved in a bridge, so the time penalty in potentially re-computing the implications of a gate more than once is far less than computing and storing the implications of all gates (most of which are not involved in any bridging fault) upfront.
b) We might lose some information in terms of constants (a line that is uncontrollable to some logic *v* is said to be a constant with value *v'*), by not evaluating the implications of every gate in the circuit. However, we more than compensate for this through the sequential symbolic simulation.
c) By not performing extended backward implications, we might not have as powerful an implication engine, as we might have with the application of extended backward implications. Though this is true, the time penalty associated with the computation of extended backward implications for every unjustified gate in the implication list for a gate outweighs the advantage of a more powerful implication engine available with extended backward implications.

2.2 Symbolic Simulation:

Symbolic simulation is performed as a preprocessing step to untestable bridge analysis; it identifies nets that are uncontrollable to a specific logic value. For example, line 'b' in the circuit shown in Figure 4 is uncontrollable to logic '0' assuming that the initial state of the flip-flop is unknown, and line 'a' is fully controllable to any value.

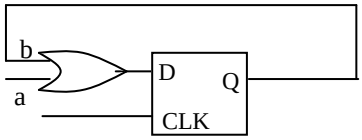


Figure 4: concept of uncontrollable net

We perform symbolic simulation using 11 symbols, that represent the controllability charactersitics of any given line. Node "*G* is uncontrollable to a logic value *V*" means:

There is no single finite sequence *I*, which when applied to the controllable points in the net-list, can set *G* to value *V* for every possible power up state of the sequential elements in the design, where $V \in \{0,1,Z\}$.
'Z' in the value set refers to the high impedance state of a net. Since most of the industrial circuits have tri-state devices and bus structures, we support the high impedance value in our tool.

The symbols used, and their meaning is described below:

1) Strong_0 (Strong_1): If a line can only take a value of 'strong 0' ('strong 1'), then it is associated with this symbol. For example, a net directly connected to VSS (VDD) would take this symbolic value. (*Represented as ST_0 (ST_1)*).
2) ControllableToAny (*C_ANY*) : If a particular net can assume any value from the defined value set, then it is associated with the 'ControllableToAny' symbol. This symbol is associated with all the primary inputs and the scan elements.
3) UncontrollableTo_0 (UncontrollableTo_1): A net that cannot be controlled to logic 0 (logic 1) is associated with

the 'UncontrollableTo_0' (UncontrollableTo_1) symbol. For example, the output of the OR gate shown in Figure 4 cannot be driven to logic 0, and hence would assume this symbol. (*Represented as UN_0 (UN_1)*).

4) UncontrollableToAny (*U_ANY*): This symbol is asociated with the net that cannot be controlled to any value. This is the symbol associated with the output of every non-scan latch/flip-flop at the beginning of symbolic simulation.

The symbolic values described above can be associated with any gate/net in the circuit, while the ones described below can only be associated with the nets that produce/transmit a high impedance (Z) state.

5) Strong_Z : Associated with the net that can only take the high impedence state. For example, a tri-state device with an active high enable would assume this symbolic value, if the enable line is a constant 0.

6) UncontrollableTo_Z: A net would assume this value if it can never achieve the high impedence state. For example, a controlled buffer with an active high enable would be associated with this symbolic value if the enable line is a constant 1.

7) UncontrollableTo_01: If a net can never be driven to either 0 or 1, then it would be associated with this symbolic value.

8) UncontrollableTo_1Z: A net which cannot achive either a high impedenece state or cannot be excited to logic 1 would be associated with this symbolic value. For example, a wired-AND gate (as shown in Figure 5) with one driver set at "UncontrollableTo_1Z" would be associated with this symbolic value, as the "UncontrollableTo_1Z" on one of the driver's o/p would dominate the value on any other driver.

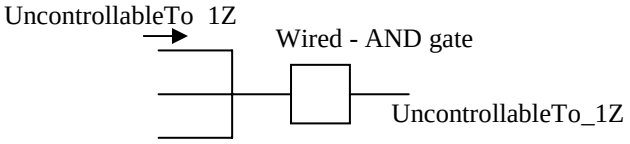


Figure 5: Wired-AND gate gets associated with symbolic value of UncontrollableTo_1Z.

9) UncontrollableTo_0Z: A net which cannot be excited to either a logic value of 0 or Z would be associated with this symbolic value. For example, a wired-OR gate with one driver as "UncontrollableTo_0Z" would be associated with the symbolic value of UncontrollableTo_0Z (as shown in Figure 6).

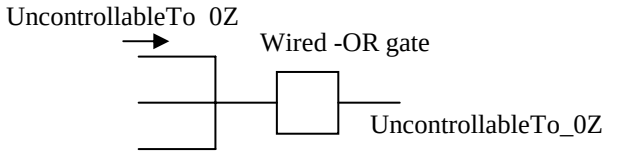
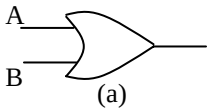


Figure 6: Wired-OR gate gets associated with symbolic value of UncontrollableTo_0Z.

Symbolic simulation is initiated by assigning *ControllableToAny* symbol to all primary inputs and scan elements, and *UncontrollableToAny* symbol to all latch elements/flip flops. The simulation is event-driven; it proceeds in a levelized manner, and each gate is evaluated one time frame after another, until the circuit settles down to a stable state. To propagate symbols, pre-computed characteristic propagation table for each gate is looked up to determine the resulting symbolic value for the gate. Such a characteristic table for a 2-input OR gate is shown in Figure 7.



(a)

A \ B	C_ANY	U_ANY	ST_0	ST_1	UN_0	UN_1
C_ANY	C_ANY	UN_0	C_ANY	ST_1	UN_0	C_ANY
U_ANY	UN_0	U_ANY	U_ANY	ST_1	UN_0	U_ANY
ST_0	C_ANY	U_ANY	ST_0	ST_1	UN_0	UN_1
ST_1	ST_1	ST_1	ST_1	ST_1	ST_1	ST_1
UN_0	UN_0	UN_0	UN_0	ST_1	UN_0	UN_0
UN_1	C_ANY	U_ANY	UN_1	ST_1	UN_0	UN_1

(b)

Figure 7: (a) OR gate (b) Its symbolic evaluation table

2.3 Fault Models

The fault models that we use to describe the behavior of a bridge, along with conditions that are required to excite the bridge fault of each type, is shown in Table 1[7]. Column one in Table 1 represents the bridge fault models used. Column two shows the representation for each model and column three shows the excitation condition(s) for each bridging fault type. Here, *a* and *b* represent the nets in the good machine, while *a'* and *b'* represent the same nets in the faulty machine. Also, *a* DOM *b* implies that logic values on net *a* dominate that on net *b*, while *a* DOM*V b* means that net *a* dominates *b* only if *a* = *V*. It may be argued that since the excitation conditions for the DOM bridge fault covers the excitation condition for the DOM0 and DOM1 bridge fault, analyzing DOM0 and/or DOM1 bridge fault model between two nets would not be necessary if the DOM bridge fault between the nets is being analyzed. This would be true if the DOM bridge fault between the two nets is found to be untestable. Then both DOM0 and DOM1 bridge faults would also be untestable, and it would be redundant to analyze them. However, the converse is not true. That is, it is possible that even though a DOM bridge between two nets is testable, either DOM0 or DOM1 bridge fault could still be untestable. So, it would be necessary to consider the DOM0 and DOM1 fault models even though they form a subset of fault models such as DOM or Wired-AND or Wired-OR model.

3. Algorithm

The following definition of an observation point is necessary to understand before we can discuss the algorithm:

Definition 1: (Observation point)

An observation point for a fault is defined as the nearest fanout stem, a primary output or a latch element, in the fanout cone of the fault site.

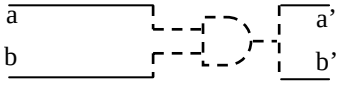
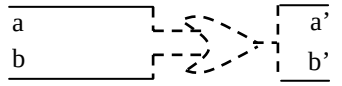
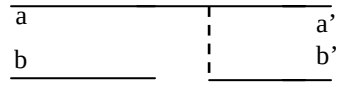
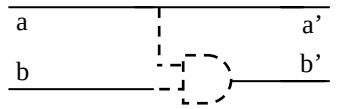
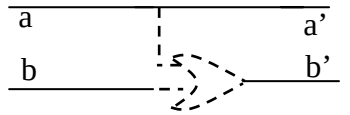
Fault Model	Representation	Excitation Condition(s)
Wired-AND Bridge (a AND b)		1) (a = 0) and (b = 1) 2) (a = 1) and (b = 0)
Wired-OR Bridge (a OR b)		1) (a = 0) and (b = 1) 2) (a = 1) and (b = 0)
DOM Bridge (a DOM b)		1) (a = 0) and (b = 1) 2) (a = 1) and (b = 0)
DOM0 Bridge (a DOM0 b)		1) (a = 0) and (b = 1)
DOM1 Bridge (a DOM1 b)		1) (a = 1) and (b = 0)

Table 1: Fault models, their representations and excitation condition(s)

Also, our analysis is based on the ILA (iterative logic array, of fixed length K) representation of a sequential circuit, where the latch elements of the lowest time frame (i.e.frame –K/2) of the ILA are fully controllable, and the latch elements in the right most time frame (i.e. frame K/2) are fully observable.

Figure 8 outlines the overall algorithm:

a) Perform Symbolic Simulation to obtain information about constants in the circuit.
Absorb the untestable stuck-at information to screen out some bridging faults.
b) For the remaining faults (per-fault analysis):
  b.1) Find the necessary condition(s) to excite the fault, and to propagate the fault effect to an "Observation Point".
  b.2) Backward imply/justify these set of conditions.
  b.3) If no conflict exists, forward imply with the logic values set in the circuit. If a conflict exists in backward / forward implications, bridge fault is untestable.
  b.4) Perform observability analysis to determine if the fault effect can be propagated to a PO or to the latch/flip-flop boundary in the last time frame of the k-frame unrolled circuit.

Figure 8: Algorithm for untestable bridge identification

Let us now look at the algorithm in detail:

a) Symbolic simulation and absorption of stuck-at information: As discussed earlier in section 2.1, we perform a sequential symbolic simulation using 11 symbols, to identify controllability characteristics of nets in the circuit. At the beginning of the simulation, we associate the primary inputs (PIs) and the controllable latch elements with the symbol *ControllableToAny*, and the other sequential (latch) elements with the symbol *UncontrollableToAny*, and propagate these symbols across the circuit. We use the table look-up scheme described earlier, until the circuit stabilizes to steady state w.r.t. symbolic values.

We absorb a fault list which lists all untestable stuck at faults in the circuit, and screen out certain bridge faults before starting the per-fault analysis. For example, if the fault "*b* stuck-at 0" is untestable, then the bridging fault "*a* DOM0 *b*" between nets *a* and *b* would also be untestable, and can be dropped from the original bridge fault list. Our tool that identifies untestable stuck-at faults is based on symbolic simulation and Boolean satisfiability analysis, but we would not discuss the details of that tool in this paper.

b) Per-Fault Analysis: After filtering the initial fault list, we perform the following analysis on each fault in the reduced list:

b.1) Identification of conditions necessary for fault detection: Depending upon the fault type, we identify the fault site and the condition(s) for fault excitation (as previously shown in Table 1). For example, if a "a DOM0 b" bridge exists between nets *a* and *b*, then the fault excitation condition would be (a = 0) and (b = 1), and the fault site would be net *b*, because a fault effect of 0/1 (good machine value / faulty machine value) appears on this net. Then, we identify the conditions necessary to propagate the fault effect to an observation point. This basically means that we identify the non-controlling values on each gate in the off-path from the fault site to the observation point. These conditions *must* be met if the fault has to be testable.

b.2) A backward implication routine is then invoked on the necessary set of conditions. We could have separately implied each necessary condition and added the individual implication lists to obtain the composite implication list corresponding to the set of necessary conditions. This is exactly what would have been done if implications corresponding to each gate were computed as a pre-processing step. However, we compute the implications associated with the entire set of necessary conditions in one go, because this technique helps in learning more implications. The backward implication process begins at the fault injection frame, and goes on till the lowest time frame in the ILA expansion is reached (i.e. –K/2, in an ILA representation of K frames) or till no more implications can be learnt.

b.3) If no conflict is encountered in the backward implication process, then a forward implication process is invoked on the already learnt implication set. This implication process begins at the lowest level of the ILA expansion, and continues till the last frame of the ILA is reached (i.e. begins at –K/2, or the lowest frame that has a valid implication associated with it, and continues till frame K/2) or till no more implications can be learnt.

b.4) If no conflict occurs during either the backward implication process or during the forward implication process, then an observability analysis is carried out to determine if the fault effect can be observed or not. This observability analysis marks the last step of untestability analysis for a fault, and the motivation behind the observability analysis is to check if the excitation conditions for the fault block the fault effect from propagating to the primary output(s) or not. The way we implement this observability analysis is through multiple ID propagation. This is explained below:

We associate an ID with the fault site, (say X), and propagate it through the circuit. We also keep a track of the polarity of this ID, and identify the inverted value of this ID as X'. This is done to identify masking of fault effects (For example, if an id of X and X' appear at the input of an AND gate, the fault effect would get killed due to the opposite polarities of the fault IDs). Since most of the industrial designs contain tri-state devices, and buses, we use another ID, which is a generic ID, and we call it IDG. The motivation of using IDG can be understood through Figure 9.

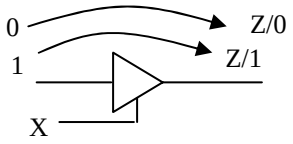


Figure 9: Use of generic ID, IDG

Let's assume that fault ID X (lets assume X represents a faulty state of 0/1) appears at the select line of a tri-state device, as shown in figure 9, with an active high enable. The faulty machine value at the output of the gate would be the value applied at the input of the gate, but the good machine value will be Z, i.e. high impedance state. A fault effect of Z/0 or Z/1 cannot be directly associated with either X or X' (because X and X' are of the form 0/1 or 1/0). Thus, this fault effect is represented by a new ID, i.e. IDG.

Thus, if any of these IDs, i.e. X, X' or IDG propagate to either a PO or to the latch boundary in the last time frame of the ILA expansion, then the fault is not declared untestable. However, if none of these IDs can propagate to a completely observable point, then the bridge is marked as untestable.

4. Results:

The proposed algorithm was implemented in C++ and experiments were conducted on ISCAS89 and some industrial circuits on a 500 MHz, HP workstation. The results are reported in Table 2. Column 1 in Table 2 shows the ISCAS '89 circuits, and the industrial circuits (C1, C2, C3) for which the tool was evaluated. Column 2 shows the number of faults our bridge fault ATPG engine identified as untestable, with the backtrack limit set to 1000. Column 3 shows the number of faults identified as untestable using the new implications-based tool, with the time frame expansion of the circuit being from –2 to 2. The numbers in the parenthesis in this column show two values. The first value is the number of faults identified as untestable both by the new tool and by ATPG, and the second value in bold identifies the number of faults that our tool identified as untestable, for which ATPG aborted after 1000 backtracks. For example, for circuit s5378, ATPG identified 261 faults as untestable. With the implications based approach, we identified 552 faults as untestable. Out of these 552 faults, 187 faults were identified as untestable by both ATPG and the implications based tool, while the rest 365 faults identified as untestable, were the faults on which ATPG aborted. So, the gain here is w.r.t the 365 faults which our new tool could identify as untestable, while for which ATPG could not generate a pattern. Column 5 shows the number of faults identified as untestable by the new tool, using K = 0, i.e. by unrolling the sequentail circuit in just one time frame (effectively combination analysis). Finally, columns 4 and 6 show the time taken by the implications based tool to analyse all faults for an ILA size of 5 (K = 2) and ILA size of 1 (K = 0) respectively. Table 3 shows the time spent by ATPG to analyze the faults identified as untestable using the implications based algorithm. Column 1 in Table 3 shows the circuit name, and the 2nd column shows the time spent by ATPG only on the faults which were identified as untestable using the implications based tool (S_{unt} for K = 2). For example, for s15850, the ATPG engine took 953 seconds to analyze the 1190 faults identified as untestable by our tool for K = 2. It was observed that the number of untestable faults identified with K = 0 (combinational analysis) is almost the same as the number of faults identified by expanding the ILA to a length of 5 frames for most of the circuits. Thus, it can be seen that even in a combinational framework, our tool can quickly identify more untestable faults than the ATPG tool can capture. It was also seen that although our implications based approach for identifying untestable bridges worked out better than ATPG for ISCAS circuits, it identified only a subset of the faults identified as untestable by the ATPG engine for the

Circuit	ATPG Red. (bk = 1000)	S_{unt} (K* = 2)	Time (sec) K = 2	S_{unt} (K = 0)	Time (sec) K = 0
s5378	261	552 (187/**365**)	361	176 (176/**0**)	40
s9234	965	5585 (679/**4906**)	356	5568 (676/**4892**)	47
s13207	1595	13163 (1485/**11711**)	318	13160 (1485/**11708**)	31
s15850	496	1190 (230/**960**)	804	1183 (228/**955**)	95
s35932	1382	945 (915/**30**)	979	927 (897/**30**)	534
s38417	535	428 (115/**311**)	2139	418 (111/**307**)	100
s38584	1113	868 (714/**154**)	3085	861 (711/**150**)	846
C1	154	39 (39/**0**)	63	39 (39/**0**)	23
C2	43	27 (27/**0**)	90	27 (27/**0**)	63
C3	76	21 (21/**0**)	97	21 (21/**0**)	14

*K = ± number of frames in which the sequential circuit is unrolled

Table 2: Results obtained for 16000 randomly generated bridges

Circuit	T_{ATPG} (sec)
s5378	58
s9234	592
s13207	14380
s15850	953
s35932	60
s38417	86
s38584	405
C1	10
C2	13
C3	8

Table 3: Time taken by ATPG to analyze S_{unt}

industrial circuits. This is primarily due to the fact that the bus structures present more commonly in industrial designs limit the strength of implications, and hence limit the untestable fault identification capability. Nevertheless, our tool provides a novel technique to quickly identify non-trivial untestable bridging faults, thereby reducing the size of the bridging fault list to be analyzed by bridge-fault ATPG engines.

5. Conclusion:

In this paper, we proposed an implication based approach complemented with a symbolic simulator to identify untestable bridging faults at gate level. Our tool was tested on large ISCAS '89 circuits, and three industrial circuits. We found the following:

a) The proposed algorithm could identify a lot of untestable bridging faults in ISCAS '89 circuits.
b) The proposed algorithm when applied to an ILA of size K = 0 could identify almost as many untestable faults as could be identified with an ILA of size 5.
c) Our tool is very inexpensive in terms of both memory and the time taken to identify untestable faults. Both these performance metrics (time and memory) are always linear in the size of the circuit. Also, the faults that we identify as untestable are non-trivial, as ATPG aborts for most of these faults.

References:

[1] Hsing - Chung Liang, Chung Len Lee, Jwu E. Chen. Identifying untestable faults in sequential circuits. *IEEE Design and Test of Computers*. Pages 14-23, Fall 1995.

[2] S. T. Zachariah, S. Chakravarty, C. D. Roth. A novel algorithm to extract two-node bridges. Design Automation Conf., 2000. pages 790-793.

[3] S.T. Zachariah, S. Chakravarty. A novel algorithm for multi-node bridge analysis of large VLSI circuits. *VLSI Design, 2001*, pages 333-338.

[4] S.D. Millman, E.J. McCluskey, and J.K. Acken. Diagnosing CMOS bridging faults with stuck-at fault dictionaries. In *Int'l Test Conf*., pages 860-870, 1990.

[5] S. D. Millman, Sir J.P. Garvey. An accurate bridging fault test pattern generator. *Int'l Test Conf*., pages 411-417, 1991.

[6] P.C. Maxwell and R.C. Aitken. Biased voting: A method for simulating CMOS bridging faults in the presence of variable gate logic thresholds. In Int'l Test Conf., pages 63-72, 1993.

[7] M. Abramovici, M.A. Breuer and A.D. Friedman. *Digital Systems Testing and Testable Design*. Cmputer Science Press, 1990.

[8] M. A. Iyer and M. Abramovici, "FIRE: a fault independent combinational redundancy algorithm", *IEEE Trans. VLSI sy.s, June 1996, pages. 295-301*.

[9] M.A. Iyer, D.E. Long, Abramovici, "Identifying sequential redundancies without search," *Design Automation Conference, 1996, pages. 457-462.*

[10] J. Zhao, J. A. Newquist and J. Patel, "A graph traversal based framework for sequential logic implication with an application to c-cycle redundancy identification", *Proc. VLSI Design Conf., 2001, pp. 163-169.*

[11] V. D. Agrawal and S. T. Chakradhar, "Combinational ATPG Theorems for Identifying Untestable Faults in Sequential Circuits," *IEEE Trans. CAD, vol. 14, Sept. 1999, pp. 1155-1160.*