Sequential Redundancy Identification Using Recursive Learning

Wanlin Cao Dhiraj K. Pradhan
Department of Computer Science
Texas A&M University
College Station, Texas 77843

Abstract

A sequential redundancy identification procedure is presented. Based on uncontrollability analysis and recursive learning techniques, this procedure identifies c-cycle redundancies in large circuits, without simplifying assumptions or state transition information. The proposed procedure can identify redundant faults which require conflicting assignments on multiple lines. In this sense, it is a generalization of FIRES, a state-of-the-art redundancy identification algorithm. A modification of the proposed procedure is also presented for identifying untestable faults. Experimental results on ISCAS benchmarks demonstrate that these two procedures can efficiently identify a large portion of c-cycle redundant and untestable faults.

1 Introduction

Identifying sequentially untestable and redundant faults is a formidable yet important problem. The presence of untestable and redundant faults complicates automatic test pattern generation (ATPG). Additionally, redundancies induce many other detrimental effects [14]. Redundancies increase chip area, power consumption, and often, propagation delays in the circuit. Usually, redundancy implies inefficient design. The presence of a redundant fault may preclude detection of other faults in the circuit. Redundancy identification is very useful in synthesis, since a redundant fault defines a region that can be removed.

Using a traditional sequential ATPG tool, the search space must be exhausted to prove that a targeted fault is untestable. But even so, none of the practical state-of-the-art sequential test generators can distinguish untestable and redundant faults. Most previous methods for identifying sequentially redundant and untestable faults [4] [5] [6] are based on ATPG, requiring large amounts of computations. Recently, two fault-independent sequential redundant and untestable fault identification algorithms, FIRES [13] and FUNI [3], have been proposed. Based on uncontrollability and unobservability analysis, FIRES finds redundant faults whose activation or observation requires conflicting assignments on fanout stems. FUNI identifies untestable faults, whose detection requires illegal states as a necessary condition. Here, by illegal states, it is meant states that cannot be entered from an unknown state. Unlike ATPG-based algorithms, these sequential implication-based algorithms do not need to perform exhaustive search; thus, they can efficiently find a large portion of redundant or untestable faults.

A new algorithm for identifying sequential redundant faults is proposed here. This algorithm uses a recursive learning-improved implication. Recursive learning has been shown to be quite useful for identifying combinationally redundant faults [11]. Similar to FIRES, our algorithm identifies sequential redundant faults by detecting conflicts in the necessary state assignments. The proposed algorithm performs sequential implication on every flip-flop, and checks the conflicting states necessary for the detection of faults. It can identify the conflict requirement on either a fanout stem or on multiple lines. In this sense, the proposed algorithm can be thought as a generalization of FIRES. Also presented is a modification of the proposed algorithm which can efficiently identify sequential untestable faults. Experimental results depict that for some circuits, the proposed algorithm outperforms previously reported results.

The paper is organized as follows. Section 2 reviews the FIRES and FUNI algorithms, and recursive learning technique. Section 3 describes the proposed algorithm. Section 4 presents experimental results, and conclusions are presented in Section 5.

2 Preliminaries

This Section introduces the terminology used in the paper, and provides a brief review of previous work on sequential untestable and redundancy identification (RID) and recursive learning technique. For RID, only techniques that are based on uncontrollability and unobservability analysis are reviewed. A detailed review of other RID techniques can be found in [14].

This work is supported in part by NSF under Grant MIP 94-06946 and byONR under Grant N00014-92-1-1366.

ICCAD '96
1063-6757/96 $5.00 © 1996 IEEE

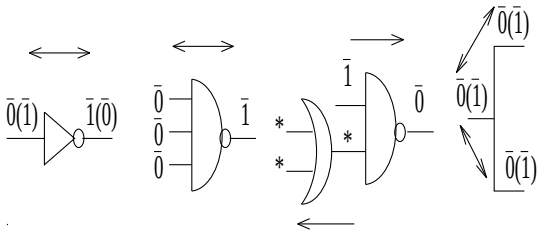

Figure 1: Propagation of Uncontrollability and Unobservability

2.1 Identifying Combinational Redundancy

Let $\{X = a, Y = b, Z = c\}$ be an illegal combination of values in a combinational circuit. Faults for which this combination of values is necessary for detection are redundant. Let S_x, S_y and S_z be the sets of faults that require $X = a, Y = b$ and $Z = c$, for detection, respectively. Because the conditions for S_x, S_y and S_z cannot be satisfied simultaneously, faults in $S_{xyz} = S_x \bigcap S_y \bigcap S_z$ are redundant.

The faults that belong to S_x can be found by uncontrollability and unobservability analysis [1, 2, 3]. Let $\overline{0}(\overline{1})$ denote the status of a line that is uncontrollable to value 0(1). Figure 1 illustrates the uncontrollability propagation rules. If a gate input cannot be set to the uncontrolling value of the gate, all the other inputs of the gate become unobservable (denoted by *). Uncontrollability propagates forward and backward, while unobservability only propagates backward.

A general procedure to find the redundant faults caused by an illegal combination $\{l_1 = v_1, l_2 = v_2, ...\}$ is:

1. For every $l_i = v_i$, imply $l_i = \overline{v_i}$ to determine all lines becoming uncontrollable or unobservable. Let S_i be the set of corresponding faults. (Note that the implications are performed separately for each value.)
2. The redundant faults caused by a given illegal combination are in the set $S = \bigcap_i S_i$.

In [10], a combinational redundant fault identification algorithm, FIRE, was proposed. FIRE finds combinationally redundant faults by performing $\overline{0}$ and $\overline{1}$ analysis on every fanout stem. If a line is set to $\overline{0}(\overline{1})$ by the propagation of both $\overline{0}$ and $\overline{1}$ on a stem, then a s-a-1 (s-a-0) fault on this line is redundant. Both faults on a line marked as unobservable by propagating both $\overline{0}$ and $\overline{1}$ are redundant.

2.2 Identifying Sequentially Redundant and Untestable Faults

In sequential circuits, untestability and redundancy are two different concepts. Several different definitions for sequential untestability and redundancy can be found in literature; this paper uses the definitions contained in [5].

Definition 1: A fault *f* is said to be *detectable* if there exists an input sequence I such that for every pair of initial states S and S^f of the fault-free and faulty circuit, respectively, the response $Z(I, S)$ of the fault-free circuit to the input sequence I is different from the response $Z^f(I, S^f)$ of the faulty circuit (at some time on some output).

Definition 2: A fault is *untestable* if it is not detectable.

Definition 3: A fault is *partially testable* if there exists an initial state S^f of the faulty circuit and an input sequence I such that for every fault-free initial state S, the response of the fault-free circuit to I, starting from $S, Z(I, S)$, is different from the response of the faulty circuit starting from $S^f, Z^f(I, S^f)$. (This definition differs from that in [5], in that it includes testable faults. A fault is testable if it is partially testable for all initial states of the faulty circuit.)

Definition 4: A fault is *redundant* if it is not partially testable.

The concept of c-cycle redundancy [14] is a generalization of the conventional redundancy defined above.

Definition 5: Consider the set of states $\{S_c\}$ reachable after powering up the faulty circuit and applying any sequence of c input vectors. (Note that $\{S_c\}$ shrinks as c increases.) Then a fault is $c-$cycle redundant if it is not partially testable under the assumption that the initial states of the faulty circuit are restricted to $\{S_c\}$.

In the following discussion, the iterative array model is used to represent the sequential circuit. The combinational noncontrolling and unobservability analysis [9, 10] is extended to sequential circuits by adding the flip-flop propagation rule. When uncontrollability/unobservability propagates forward (backward) through a flip-flop, it enters the next (previous) time frame. In sequential circuits, unobservability propagation is different from that of combinational circuits. Before propagating unobservability from the output of a gate to all its inputs, it must be ensured that multiple fault-effects from that input, in different time frames, cannot reach a primary output. Otherwise, faults that can be detected by (sequential) multiple path sensitization would be marked as unobservable.

In [14], a sequential redundancy identification procedure, FIRES, was proposed. FIRES applies sequential uncontrollability/unobservability on every fanout stem and finds c-cycle redundant faults for which a conflicting value assignment on the stem is necessary for detection. Since a fault is present in every time frame, a validation procedure is used to guarantee that the uncontrollability and unobservability propagation is not invalidated by the fault. The process of propagating uncontrollability and unobservability to determine the uncontrollable and unobservable faults is referred to as *sequential implication*. In [8], it is proven that a fault identified by FIRES is c-cycle redundant.

In [3], FUNI, a sequential untestable identification algorithm, was proposed. FUNI is composed of two steps. First, an implicit state enumeration algorithm, based on the

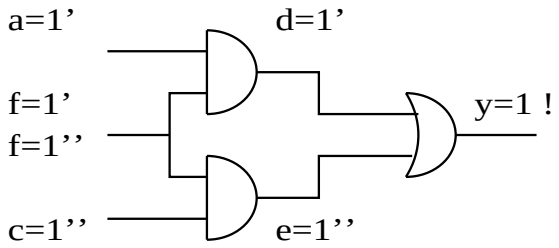

Figure 2: Recursive Learning Example

Binary Decision Diagram (BDD) [15], is used to identify illegal states. Then, for every illegal state, the sequential implication procedure is applied to find an untestable fault, by showing that its detection requires an illegal state as a necessary condition. To reduce the complexity of creating BDD and the implicit state enumeration process, FUNI adopts a function-partitioning technique to divide the problem. But for some circuits, because the size of BDD can be exponentially large, the memory requirement may still pose a problem.

FUNI only identifies untestable faults; it does not need the validation procedure in FIRES [14]. FUNI first finds illegal states, then analyzes every illegal state to determine the faults whose detection requires the fault-free circuit to enter that state. This runs the risk of analyzing many illegal states that are “harmless", that is, they are not necessary for detecting any faults. In contrast, we first determine the faults whose detection require a certain state; then we use recursive learning to attempt to justify that state. If the justification fails, the state is illegal, and the faults requiring it are untestable.

2.3 Recursive Learning

ATPG search attempts to justify a given value assignment by searching for other node value assignments necessary to maintain the logic consistency of the circuit. These are known as *necessary assignments* For example, in order to justify a 1 at the output of an AND gate, all of its inputs must be assigned the value 1. In standard ATPG search, these assignments are discovered by making *direct implications*. A direct implication is a logical consequence of the truth table for the logic function. For example, in the case of a 1 on an AND gate, the only entry in the truth table for which it is true for all inputs assigned the value 1. *Indirect implications* are logical consequences of that do not directly follow from the logic function truth table, but are caused by the circuit structure.

An example is shown in Figure 2. If the assignment y = 1 is made, it is not possible to make any direct implication about values a, c, d, e, or f. However, there exists the indirect implication that $y = 1 \Rightarrow f = 1$. This can be discovered by using recursive learning [11, 12]. The outline of the algorithm is shown in Figure 3.

```
r = 0
make_all_implications(r,r_max) {
  make all direct implications
  set up list U_r of all unjustified functions
  if r < r_max
    for each function f_i in U_r
      set up list of justifications C_r^f
      for each justifications J_i in C_r^f
        make assignments in J_i
        make_all_implications(r+1,r_max)
      if there is one or several signal f in the circuit, which
      assumes same logic value V for all consistent justifications
      J_k in C_r^f then learning y_i = V_i is uniquely
        determined in level r
        make direct implications for all y_i = V_i in level r
      if all justifications are inconsistent
        learn given situation of value assignments in level r
        is inconsistent
}
```

Figure 3: Recursive Learning Algorithm

Beginning at the point where direct implication cannot be made (y, in this case), temporary values are injected into the circuit for each choice of value assignments that would justify the current node values. In this case, there are two choices: setting either d = 1 or e = 1. The temporary assignment is made, and then direct implications are made to set all necessary assignments; this is repeated for all choices. In the example, the two choices and their necessary assignments are marked with prime and double prime. If there is an intersection of the necessary assignments for each choice, this becomes a necessary assignment for all choices. In the example, f = 1 for both d = 1 and e = 1. Given the existence of the indirect implication $y = 1 \Rightarrow f = 1$, the ATPG search can immediately proceed to set f = 1, which eliminates half the input search space. If, in turn, f is fed by an OR gate, then the learning procedure can *recurse*, attempting to discover further implications; thus, the name *recursive learning*. The number of recursions is referred to as the *learning level* or *recursion level*. Since the number of temporary value assignments rises exponentially with the recursion level, recursive learning is operated with a maximum recursion level, to limit the amount of learning performed.

The time complexity is determined by the number of levels of recursions and fan-ins. However the maximum number of recursion levels is bounded by the levels of logic. While it is true that time complexity is exponential in terms of levels of recursion, the memory requirement is linear, in terms of number of gates, as illustrated below.

For the circuit in Figure 4, consider the primary output, $t = 1$. We can invoke recursive learning to determine any indirect implications of this value assignment. The value $t = 1$ makes the gate G_7 unjustified. We enter level 1

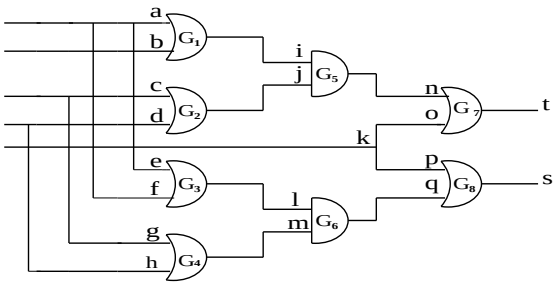

Figure 4: Levels of Recursive Learning

recursion with two possible justifications : $J_1 = \{n = 1, o = x\}$ and $J_2 = \{n = x, o = 1\}$. For $n = 1$, we obtain $i = j = 1$ through direct implication. With $i = 1$ and $j = 1$, we enter the second level of recursion, and obtain two sets of justification. For gate G_1, $J_3 = \{a = 1, b = x\}$ or $J_4 = \{a = x, b = 1\}$ are the two possible justifications. Through direct implication, we deduce $l = 1$. Similarly with level 2 recursion, we can deduce that $G_2 = 1$ implies $G_4 = 1$. Thus, by direct implication, we obtain that $t = 1$ implies $q = 1$. Similarly, with $o = 1$, we have $p = 1$. In either case, we have $s = 1$, which is the consequence of the intersection implications, resulting from the two possible justifications, J_1 and J_2, at gate G_7. It can be seen that the intermediate of implications, such as $G_5 = 1$ implying $G_6 = 1$, are not learned because the only value assignment at hand is $t = 1$. The indirect implications learned can be very useful in ATPG. Given enough recursion level, it can eliminate all backtracks and can be used to identify redundant faults [11]. It may also be noted that unlike other learning methods, such as functional learning [16], recursive learning is self-guided. The algorithm guides itself to those parts of circuit where useful learning can occur.

3 The Proposed Algorithm

This section first introduces a c-cycle redundancy identification algorithm. Then it is shown how it can be modified to efficiently find sequential untestable faults.

3.1 Identifying c-Cycle Redundancy

The proposed redundancy identification algorithm (*Procedure 1*) is as illustrated in Figure 5. *Procedure 1* is composed of two steps. The first step performs sequential $\overline{0}$ and $\overline{1}$ analysis on every flip-flop. For every line, a 0-list and a 1-list is computed. The $0(1)$ list of line l stores the flip-flop assignments that are necessary to set line l to $0(1)$. If the effect of a flip-flop's sequential uncontrollability propagation reaches a line and makes it $\overline{0}(\overline{1})$, then the flip-flop's current value is referred to in the line's 0(1)-list. If a line becomes unobservable, then the flip-flop's current value is referred to in both its 0- and 1-lists.

For the circuit shown in Figure 6, Table 1 illustrates some lines' 0 and 1-lists after performing $\overline{0}$ and $\overline{1}$ implications on the two flip-flops, f and h. It is assumed that the sequential uncontrollability analysis process begins at time frame 0.

Identifying_Redundancy (T_m)
/* T_m = Maximum # of time frames */
/*l^i means line l in time frame i */
Step 1:
For every flip-flop F_i {
Sequential Imply $F_i = \overline{0}$ over T_m time frames;
If a line l^i becomes $\overline{0}(\overline{1})$
then keep $F_i = 0$ in line l^i's $0(1)-$ list;
If a line l^i becomes unobservable
then keep $F_i = 0$ in line l^i's 0-list and 1 list;
Sequential Imply $F_i = \overline{1}$ over T_m time frames;
If a line l^i becomes $\overline{0}/\overline{1}$
then keep $F_i = 1$ in line l^i's 0/1 list;
If a line l^i becomes unobservable
then keep $F_i = 1$ in line l^i's 0-list and 1 lists;
}
Step 2:
For every line l^i **within** T_m **time frames** {
If line l^i's 0-list or 1-list has more than one flip-flops' value assignments {
Get state S_j (kept in l^i's 0-list or 1-list);
Inject S_j values into the circuit;
Inject fault l^i s-a-1 or s-a-0 into all the time frames $j < i$ (for validation purpose);
Check current value assignments consistency by using recursive learning;
}
}

Figure 5: *Procedure 1* – Identifying Redundancies

After step one, each line's 0-list and 1-list stores some flip-flops' value assignments or a state. The second step of this algorithm checks whether such a state is consistent. This is achieved by injecting the state into the circuit and performing logic implication. If a conflict occurs, then it is known that the state is inconsistent. For a line *l*, if the state stored in its 0-list (1-list) is inconsistent, then fault *l* s-a-1 (s-a-0) is untestable [6, 8].

The uncontrollability and unobservability propagations are valid only for the fault-free circuit. In general, they may not be valid in a faulty circuit. The validation of the uncontrollability analysis is fault dependent. In *procedure 1*, the uncontrollability analysis step (Step 1) gives the condition for detection a fault; and a recursive learning-based implication is used to check the consistency of the condition (a state). When a fault is identified, the following validation check is performed: first, injecting the fault into all the time frames that are involved in the above analysis steps; second, performing the analysis again under the presence of the fault. In [8], it is proven that, with the validation check, all the faults identified by *procedure 1* are c-cycle redundant.

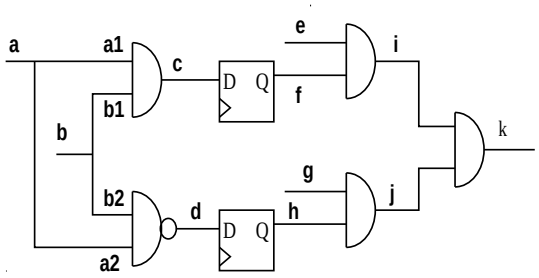

Figure 6: Find Conflict State by Direct Implication

For the circuit in Figure 6, if state $\{f = 1, h = 1\}$ is injected into the circuit and then logic implications are performed, a conflict occurs. Since state $\{f = 1, h = 1\}$ is referred to in the 0-lists of lines $a2, b2, e, g$ and in the 1-lists d, h, i, j, k, it is known that faults $a2, b2, e, g$ s-a-1 are untestable, and that faults d, h, i, j, k s-a-0 are untestable. In fact, this example demonstrates the chief difference between *Procedure 1* and FIRES. FIRES performs sequential implication on one stem at a time. So it will not identify untestable (c-cycle redundant) faults for which conflicting value assignments on multiple lines are required. For the circuit in Figure 6, FIRES can only identify faults $a2$ and $b2$ s-a-1 as untestable (c-cycle redundant).

time frame	0-list	1-list
-1	$c : \{f = 0, h = 1\}$ $a1 : \{f = 0, h = 1\}$ $b1 : \{f = 0, h = 1\}$ $b2 : \{f = 1, h = 1\}$ $a2 : \{f = 1, h = 1\}$	$c : \{f = 0, h = 1\}$ $d : \{f = 1, h = 1\}$
0	$e : \{f = 1, h = 1\}$ $g : \{f = 1, h = 1\}$ $f : \{f = 0, h = 1\}$ $i : \{h = 1\}$ $j : \{f = 1\}$ $h : \{f = 1, h = 0\}$	$e : \{f = 1, h = 1\}$ $g : \{f = 1, h = 1\}$ $h : \{f = 1, h = 1\}$ $i : \{f = 1, h = 1\}$ $j : \{f = 1, h = 1\}$ $k : \{f = 1, h = 1\}$

Table 1: Example: 0-list, 1-list

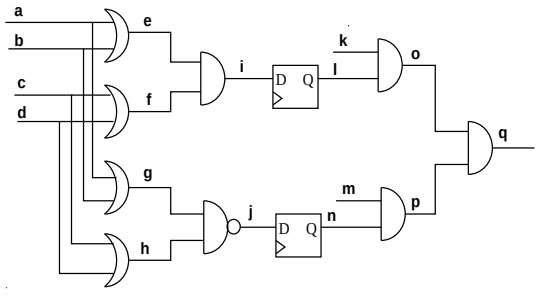

Figure 7: Find Untestable Faults By Recursive Learning

The traditional 3-valued logic implication will not find most conflicts caused by a set of value assignments. In our algorithm, a recursive learning [11-13] -based logic implication procedure is used to find more inconsistent states. For the circuit shown in Figure 7, direct logic implication will not identify conflict value assignments $\{l = 1, n = 1\}$. Using recursive learning analysis, one can determine that $l = 1$ implies $n = 0$. Therefore, the inconsistent state $\{l = 1, n = 1\}$ will be identified.

3.2 Identifying Untestable Faults

The previous Section introduced a c-cycle redundancy identification procedure. It can find redundant faults for which conflict value assignments on multiple lines are required for detection. Removing the validation step from the procedure, it can also be used to find untestable faults. This Section introduces an alternative implementation of the above idea for identifying untestable faults. Figure 8 summarize the procedure (*Procedure 2*).

Identifying_Untestable(T_m)
/* T_m = Maximum # of time frames */
/* l^i means line l in time frame i */
Step 0:
Recursive learning pre-processing;
Step 1:
For every flip-flop F_i {
RL-based Sequential Imply $F_i = \overline{0}$ over T_m;
If a line l^i becomes $\overline{0}(\overline{1})$
then keep $F_i = 0$ in line l^i's 0(1) -list;
If a line l^i becomes unobservable
then keep $F_i = 0$ in line l^i's 0-list and 1-list;
RL-based Sequential Imply $F_i = \overline{1}$ over T_m;
If a line l^i becomes $\overline{0}(\overline{1})$
then keep $F_i = 1$ in line l^i's 0(1)-list;
If a line l^i becomes unobservable
then keep $F_i = 1$ in line l^i's 0-list and 1-list;
}
Step 2:
For every line l^i **within** T_m **time frames**{
If line l^i's 0-list or 1-list has more than one flip-flops' value assignments {
Inject S_j values into the circuit;
Check current value assignments consistency by RL-Based implication;
}
}

Figure 8: *Procedure 2* – Identifying Untestable Faults

The first step in *Procedure 2* is a recursive learning pre-processing step, as in [13]. Although it is necessary to "pay" for the cost of this pre-processing, the following two facts make it worthwhile: first, because the entire analysis and state verification process in *Procedure 2* is performed on the fault-free version of the circuit (see [6]), the information learned in the pre-processing step is effective for the sequential implication and state verification process. Second, the iterative array model is used in the procedure.

Since all the time frames have the same structure, the inter- or intra- time frame learning results can be repeatedly used over all other time frames.

For the circuit shown in Figure 7, by performing recursive learning, indirect implication line $i = 1 \Rightarrow j = 0, i = 0 \Rightarrow j = 1, j = 0 \Rightarrow i = 1$ and $j = 1 \Rightarrow i = 0$ will be learned. The pre-processing step stores this information in lines i and j. Later, when dealing with line i or j, the implication procedure will check and use this stored implication.

The global implications learned can also be used to improve the sequential uncontrollability analysis procedure. In [8], it is pointed out that because of the incompleteness of the sequential uncontrollability analysis procedure, FUNI may miss some untestable faults. Our recursive learning-based uncontrollability analysis procedure has the potential of identifying untestable faults that will not be identified by a direct uncontrollability analysis procedure. This function has not yet been implemented.

The main cost of this procedure is in the initial pre-processing step. Since the global implications learned are very helpful or even essential to other steps in an ATPG tool, the extra time spent by our procedure is just the uncontrollability analysis and logic implications time.

4 Experimental Results

We used a prototype implementation of *Procedure 1* and *Procedure 2* to identify c-cycle redundant and untestable faults in the ISCAS 89 benchmark circuits. Table 2 presents a comparison with the c-cycle redundant faults identified by FIRES, and the untestable faults identified by FUNI. In Table 2, "# Unt" presents the number of untestable faults found, and "# Red" presents the number of c-cycle redundant faults found. Under "# Unt", "*Proc 2*" gives the total number of untestable faults identified by *Procedure 2*; "*Proc* 2^*" gives the number of untestable faults identified by *Procedure 2* but not identified by FUNI and FIRES. For the smaller circuits, since FUNI can find most illegal states, *Procedure 2* can perform no better than FUNI and FIRES. But for larger circuits, about half of the untestable faults identified by *Procedure 2* are not found by FUNI or FIRES. For the smaller circuits, the maximum time frames used are 9. For circuits S9234, S13207 and S15850, the maximum time frames used is 4.

Our results demonstrate that the proposed redundancy identification algorithm finds a large number of c-cycle redundant faults. Theoretically, our algorithm should be able to identify all the redundant faults identified by FIRES. In practice, only a small recursion level can be used in recursive learning; it will not identify some redundant faults that are identified by FIRES. *Procedure 1*'s ability depends on the recursion level of recursive learning and the number of time frames used in sequential implication. For example, for circuit s510, with recursion level 2, 4 redundant faults were identified; when the recursion level was increased to 3, 6 redundant faults were identified. [14] gives the conditions for propagating unobservability through a fanout line. It has not been implemented in our programs. When implemented, this should result in discovering of more redundant faults.

Ckts	# Red. Flt		# Unt. Flt		
Name	*Proc 1*	FIRES	*Proc 2*	*Proc* 2^*	FUNI
S344	5	0	5	0	5
S349	15	2	17	0	7
S382	2	17	2	0	13
S386	36	27	36	0	36
S444	5	11	5	0	23
S510	4	0	4	0	0
S713	17	32	19	0	76
S953	8	0	9	0	0
S1423	3	5	3	3	0
S1494	0	1	0	0	14
S5378	393	366	406	176	398
S9234	201	270	201	89	93
S13207	644	893	847	225	617
S15850	352	328	369	328	629

Table 2: Experimental Results: Comparison with FIRES and FUNI

Circuit	Direct Imp		RL Imp(L=1)		RL Imp(L=2)	
Name	# flt	*CPU*	# flt	*CPU*	# flt	*CPU*
S344	1	0.35	1	1.12	5	5.67
S349	0	0.76	17	2.24	17	10.99
S382	2	0.73	2	3.03	2	15.05
S386	25	3.98	36	9.58	36	47.26
S444	2	1.01	5	3.91	5	17.08
S510	0	3.08	0	10.15	4	33.67
S713	0	0.70	5	1.65	19	5.35
S953	0	17.26	3	65.31	9	126.75
S1423	0	5.81	0	11.62	3	57.62
S5378	266	114.91	353	274.06	406	974.02
S9234	112	95.56	201	191.31	201	800.07
S13207	726	225.50	847	610.19	*	*
S15850	47	474.86	369	1407.09	*	*

Table 3: Untestable Faults Identified with Different Recursion Levels

Table 3 shows the number of untestable faults identified by *Procedure 2* with different recursion levels. The unit of the *CPU* times is in seconds. All the experiments are performed on Ultrasparc workstations. It is clear that the recursive learning-based implication can find more inconsistent states. For circuits S13207 and S15850, only recursive learning level 1 was performed.

It may be seen that some of the limitations of FUNI are alleviated. For example, it is difficult for FUNI to determine when to stop the illegal state identification process. This problem can be compounded by the fact that not all illegal states identified by FUNI will lead to finding untestable

faults.

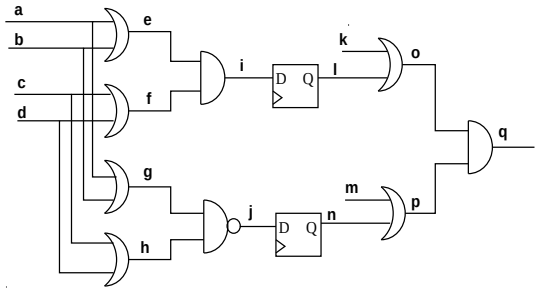

Figure 9: Example: Useless Illegal State

For the circuit shown in Figure 9, FUNI may have to spend some time to find that the states $\{f = 0, h = 0\}$ and $\{f = 1, h = 1\}$ are illegal states. But since the effect of the sequential implication of $f = \overline{0}$, $h = \overline{0}$ or $f = \overline{1}$, $h = \overline{1}$ cannot reach any common lines, these states certainly will not yield untestable faults. On the one hand, to find additional untestable faults, we need to identify more illegal states; however, the process of finding additional illegal states may be costly, as more illegal states may only lead to illegal states that do not identify any untestable faults. In extreme cases, even though no additional untestable faults could be found by new illegal states, FUNI may still try to find new illegal states.

5 Conclusions

Earlier recursive learning was shown to be useful for combinational ATPG in being able discover redundant faults in an efficient manner [11]. In this paper we show it can be quite useful in identifying redundant faults in sequential circuit as well. The proposed algorithm can be easily integrated into a synthesis system to remove redundancies. Compared with FIRES, a state-of-the-art redundancy identification algorithm, the proposed algorithm is more general and can find redundant faults which require conflict value assignments on multiple lines for detection. Also presented was a procedure to identify untestable faults. Since the main cost of this procedure is in the initial pre-processing step, and the global implications learned in this step are very helpful or even essential to other steps in an ATPG tool, the extra time spent by our procedure is simply the uncontrollability analysis and logic implications time.

In our current implementations, recursive learning is used on all the lines of the circuits. Further work in progress is on using recursive learning at higher level in the local area of the circuits. This will allow the procedures to work efficiently on the larger circuits.

6 Acknowledgement

Our special thanks to Dr. Miron Abramovici for many useful discussions and comments. We thank AT&T Bell Laboratories for the use of GENTEST, and for providing the fault lists of FIRES and FUNI. We also thank Dr. Alain Dargelas of COMPASS for using MOSA [17] to verify the untestable faults.

References

[1] M. Abramovici, J.J. Kulikowski, and R.K. Roy, "The Best Flip-Flops to Scan," *Proc. Intn'l Test Conf.*, pp.166-173, Oct. 1991.

[2] M.A. Iyer and M. Abramovici, "Sequentially Untestable Faults Identified Without Search," *Proc. Intn'l Test Conf.*, pp259-266, Oct. 1994,

[3] D.E. Long, M.A. Iyer and M. Abramovici, "Identifying Sequentially Untestable Faults Using Illegal States," *Proc. 13th. IEEE VLSI Test Symposium*,pp.4-11, May 1995.

[4] K.T.Cheng, "On Removing Redundancy in Sequential Circuits," *Proc. 28th. DAC*, pp.164-169, June 1991.

[5] I.Pomeranz and S.M.Reddy, "On Identifying Untestable and Redundant Faults In Synchronous Sequential Circuits," *12th IEEE VLSI Test Symposium*, pp.8-14, April 1994.

[6] V. Agrawal and S. Chakradhar, "Combinational ATPG Theorems for Identifying Untestable Faults in Sequential Circuits," *Proc. of Euro. Test Conf.*, Apr. 1993.

[7] H.Cho, G.D.Hachtel and F.Somenzi, "Redundancy Identification/Removal and Test Generation for Sequential Circuits Using Implicit State Enumeration," *IEEE Trans. on CAD*, vol.12,no.7, pp.935-945, July 1993.

[8] M.A. Iyer, "On Redundancy and Untestability In Sequential Circuits," *Ph.D. Thesis, Illinois Institute of Technology*, July 1995.

[9] M.A. Iyer and M. Abramovici, "One-Pass Redundancy Identification and Removal," *Proc. Intn'l Test Conf.*, pp807-815, Sept. 1992.

[10] M. Abramovici and M.A. Iyer, "Low-Cost Redundancy Identification for Combinational Circuits," *7th. Intn'l. Conf. on VLSI Design*, pp.315-318, Jan. 1994.

[11] W.Kunz and D.K.Pradhan, "Recursive Learning: A New Implication Technique for Efficient Solution to CAD Problems - Test, Verification, and Optimization," *IEEE Trans. on CAD*, vol.13,no.9, pp.1143-1158, Sept. 1994.

[12] D.K.Pradhan and W.Kunz, "Method For Circuit Verification and Multi-Level Circuit Optimization Based On Structural Implications," U.S. Patent, No. 5526514, June 11, 1996.

[13] W.Kunz, "HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning," *Proc. ICCAD*, pp.538-543, Nov. 1993.

[14] M.A. Iyer, D.E. Long and M. Abramovici, "Identifying Sequential Redundancies Without Search," *Proc. DAC 96*, June 1996.

[15] Randal E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. on Computers*, vol. c-35, no.8, August 1986.

[16] R. Mukherjee, J. Jain and D. K. Pradhan "Functional Learning: A New Approach to Learning in Digital Circuits", *Proc. IEEE VLSI Test Symp.*, pp 122-127, April 1994.

[17] A. Dargelas, C. Gauthron, Y. Bertrand, "MOSA, A Multiple Strategy Oriented Sequential ATPG", *1st European Test Workshop*, June 1996.