

# Recursive Learning: A New Implication Technique for Efficient Solutions to CAD Problems—Test, Verification, and Optimization

Wolfgang Kunz, *Member, IEEE*, and Dhiraj K. Pradhan, *Fellow, IEEE*

**Abstract**—Motivated by the problem of test pattern generation in digital circuits, this paper presents a novel technique called *recursive learning* that is able to perform a logic analysis on digital circuits. By recursively calling certain learning functions, it is possible to extract all logic dependencies between signals in a circuit and to perform *precise* implications for a given set of value assignments. This is of fundamental importance because it represents a new solution to the Boolean satisfiability problem. Thus, what we present is a new and *uniform* conceptual framework for a wide range of CAD problems including, but not limited to, test pattern generation, design verification, as well as logic optimization problems. Previous test generators for combinational and sequential circuits use a decision tree to systematically explore the search space when trying to generate a test vector. Recursive learning represents an attractive alternative. Using recursive learning with sufficient depth of recursion during the test generation process guarantees that implications are performed precisely; i.e., *all* necessary assignments for fault detection are identified at every stage of the algorithm so that *no* backtracks can occur. Consequently, no decision tree is needed to guarantee the completeness of the test generation algorithm. Recursive learning is not restricted to a particular logic alphabet and can be combined with most test generators for combinational and sequential circuits. Experimental results that demonstrate the efficiency of recursive learning are compared with the conventional branch-and-bound technique for test generation in combinational circuits. In particular, redundancy identification by recursive learning is demonstrated to be much more efficient than by previously reported techniques. In an important recent development, recursive learning has been shown to provide significant progress in design verification problems [22]. Also importantly, recursive learning-based techniques have already been shown to be useful for logic optimization. Specifically, techniques based on recursive learning have already yielded better optimized circuits than the well known MIS-II.

**Index Terms**—Recursive learning, unjustified gates, precise implications, necessary assignments, boolean satisfiability, design verification, multi-level optimization.

## I. INTRODUCTION

**T**HE PROBLEMS of test generation verification and optimization are related [22]–[26] and NP-complete [1]. Test generation for combinational circuits has been viewed

Manuscript received September 8, 1992; revised June 9, 1993. This work was supported in part by NSF and ONR. This paper was recommended by Associate Editor K.-T. Cheng.

W. Kunz was with the University of Massachusetts, Amherst, MA and the Institute für Theoretische Elektrotechnik, University of Hannover, Germany. He is now with the Max-Planck-Society, Fault-Tolerant Computing Group at the University of Potsdam, 14415 Potsdam, Germany.

D. K. Pradhan was with the University of Massachusetts, Amherst, MA. He is now with the Fault-Tolerant Computing Lab, Computer Science Department, Texas A&M University, College Station, TX 77843 USA.

IEEE Log Number 9313666.

[2] as implicit enumeration of an  $n$ -dimensional Boolean search space, where  $n$  is the number of primary input signals. Traditionally, a *decision tree* is used to branch and bound through the search space until a test vector has been generated or a fault has been proven redundant. Efficient heuristics have been reported to guide the search [3]–[5]. However, it is the nature of this classical searching technique that it is often very inefficient for hard-to-detect and redundant faults; i.e., in those cases where only few or no solutions exist.

What we propose here is a fundamentally *new* searching technique that can handle these pathological cases in test generation much more efficiently than the traditional search. It is important to note that while results presented here are in the context of test generation, they possess a wide range of applications to many important areas in computer-aided circuit design, such as logic verification and optimization [22]–[25]. Specifically, for the first time, this new searching technique provides a uniform framework for *both* CAD and test problems. The recursive learning implication procedure [18] originally developed for test problems has now been shown to provide a powerful tool for improved verification and optimization procedures. It is expected that the real potential of the framework may, indeed, be in the formulation of solutions to CAD problems. For example, using this, it has been shown [22] that the nonredundant initial MCNC versions were not equivalent to the original ISCAS 85 benchmarks—a surprising result! In addition only a few seconds on a Sparc workstation were sufficient to verify the formidable multiplier c6288 against the nonredundant version. Further application of recursive learning to design verification has recently been reported by the authors [23]. Also, the potential application of recursive learning to logic optimization and other related problems is currently under investigation by the authors, as well as others [25]. First results clearly show that recursive learning can be used to design very powerful techniques for multilevel logic optimization [25], [26]. Already, reported results indicate recursive learning-based optimization can yield smaller circuits compared to MIS-II [25], [26].

In this paper, we first apply our new approach in the context of test generation and then follow up with a discussion of to how test generation techniques can be applied to various synthesis problems. During the process of generating a test vector for a given fault, some assignments of signal values are found to be *necessary*; i.e., those assignments can be implied in anyway, and therefore they must be satisfied for the given combination of value assignments. Other assignments

are *optional* and their assignment represents a decision in the decision tree. Significant progress has been made, especially in redundancy identification, since techniques have been developed that are able to identify necessary assignments [5], [7], [8]. However, all these techniques are limited in that they *fail* to produce *all* necessary assignments. What we propose here is a technique that, for the first time, indeed generates *all* necessary assignments.

Knowledge about necessary assignments is crucial for limiting or eliminating altogether the number of backtracks that must be performed. Backtracks occur only after wrong decisions have been made that violate necessary assignments. Hence, it is important to realize that if *all* necessary (mandatory) assignments are known at every stage of the test generation process, there can be *no* backtracks at all. Since remaining assignments are only optional, they cannot therefore be violated. All methods presented in the past, such as [5]–[8], were *not able* to identify *all* necessary assignments, based as they are on polynomial time-complexity algorithms. The problem of identifying *all* necessary assignments is an NP-complete method that guarantees identifying all necessary assignments must be exponential in time complexity.

This work develops a new method called *recursive learning* that can perform a complete search to identify *all* necessary assignments. It is important to note that previous methods did not provide for this completeness. Searching for all necessary assignments provides a fundamentally new alternative to traditional techniques. For example, traditional test generation is a search for one sufficient solution, whereas recursive learning may be viewed as searching for those conditions that enables purging the nonsolution area from the search space.

One other significant attribute of recursive learning that has been successfully exploited in verification [22]–[26] and the optimization is the ability to provide all indirect implications of a particular value assignment. Such indirect implications form the basis of finding functionally equivalent nodes in verification [22]–[24]. In optimization, the recursive learning-based implication can provide a powerful tool for circuit transformations [24]–[26].

Our technique is based on performing learning operations. First introduced in [6], [7] and further developed in [12], learning is defined to mean the *temporary* injection of logic values at certain signals in the circuit to examine their logical consequences. By applying simple logic rules, certain information about the current situation of value assignments can be learned.

This work generalizes the concepts of learning in various ways: in previous learning methods [6], [7], learning is restricted to a 3-valued logic alphabet. The method presented here is not restricted to any particular logic alphabet and can be used for any logic value system such as 5-valued logic [11], 9-valued logic [9] or 16-valued logic [8], [10]. Secondly, our learning routines can be called recursively and thus provide for completeness. The maximum recursion depth determines how much is learned about the circuit. The time complexity of our method is exponential in the maximum depth of recursion,  $r_{\max}$ . Memory requirements, however, grow linearly with  $r_{\max}$ . As noted before, any method that identifies all necessary

assignments must be exponential in time complexity because this problem is NP-complete.

In broader terms, recursive learning can be understood as a general method to conduct a logic analysis, deriving a maximum amount of information about a given circuit in a minimum amount of time. This paper examines the ability of recursive learning to derive necessary assignments of fundamental importance for many applications throughout the field of computer-aided circuit design [22]–[25]. The performance of recursive learning is evaluated here by applying it to the problem of test generation. Also, results of recursive-learning-based techniques in logic verification and optimization are reviewed and summarized in this paper.

Because most state-of-the-art test generation tools, like [5], [7], [17], are further developments of the well-known FAN algorithm [3], we, therefore present and discuss recursive learning with respect to FAN-based test generation. Other approaches to test generation can also make efficient use of this new searching scheme. Other work includes an algebraic method based on Boolean difference [15] and the use of transitive closure in [16] that allows for parallelization of test generation. It should be noted that these methods, unlike our technique, rely on a decision tree when producing a complete search. Thus, they are likely to benefit from the searching scheme presented here.

This section is organized into six main sections. Section II illustrates recursive learning with an example, Section III gives a formal description of recursive learning in finding necessary assignments, Section IV discusses recursive learning for test generation, Section V gives the results for test generation using recursive learning, and Section VI briefly covers different promising applications of recursive learning to verification and synthesis problems.

## II. A SIMPLE ILLUSTRATION OF RECURSIVE LEARNING

This section introduces the recursive learning concept by first presenting a simplified (preliminary) learning technique. The basic motivation here is to illustrate concepts and show what may or may not lead to complete recognition of all necessary assignments. The formal procedure in Section III will be rooted in the observations made in the following example. Table I shows a learning procedure for unjustified lines [3], which is called recursively.

Fig. 1 shows an example to introduce the basic framework of recursive learning referred to as `demo_recursive_learning()` in Table I. For the time being, disregard the pads  $x_e, x_g, x_h, y_e, y_g,$  and  $y_h$ . These will be used later to insert additional circuitry to illustrate other key points. Consider signals  $i_1$  and  $j$ , where  $i_1 = 0$  and  $j = 1$  are two unjustified lines. Given this, the reader can derive that  $k = 1$  and  $i_2 = 0$  are necessary assignments. This is easy to observe as the nodes are labeled in a special way in Fig. 1. Signals that are labeled by the same character (excluding  $x, y$  pads), but different indices always assume the same logic value for an arbitrary combination of value assignments at the inputs. Thus, nodes labeled with the same character, except the  $x, y$  nodes, are functionally equivalent. (e.g.,  $e_1$  and  $e_2, f_1$  and  $f_2$ )

TABLE I  
PRELIMINARY LEARNING ROUTINE

```

demo_recursive_learning()
(
  for each unjustified line
  {
    for each input: justification :
    {
      - assign controlling value (e.g., '0' for AND, '1' for OR)
      - make implications and set up new list of
        resulting unjustified lines
      - if consistent: demo_recursive_learning()
    }
    if there are one or several signals f in the circuit, such that
    f assumes the same logic value V for all consistent
    justifications, then learn f=V, make implications for all
    learned signal values

    if all justifications are inconsistent: learn that the current
    situation of value assignments is inconsistent
  }
)
    
```

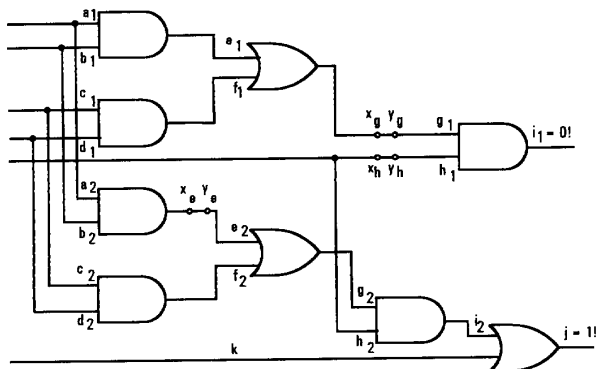


Fig. 1. Circuitry to demonstrate recursive learning.

We will now explain, step by step, how recursive learning derives the necessary assignments  $i_2 = 0$  and  $k = 1$  for the given value assignments  $i_1 = 0$  and  $j = 1$  in Fig. 1. This is done in Table II, which lists the different learning implications that occur when *demo\_recursive\_learning* is performed for the unjustified line  $i_1 = 0$ .

The first column represents the situation of value assignments before and after recursive learning has been performed. When learning is performed at the unjustified line  $i_1$  (column 2), we first assign the controlling value at  $g_1$ . Assigning the controlling value to a gate with an unjustified line represents a justification, as will be defined more generally in Section III. With the implications for  $g_1 = 0$ , we obtain the unjustified lines  $e_1$  and  $f_1$  in the first learning level. These signals are treated by recursively calling learning again (column 3 representing level 2). Now, for the unjustified line  $e_1 = 0$ , we examine the justifications  $a_1 = 0$  and  $b_1 = 0$ . In **both** these cases, we obtain  $e_2 = 0$ . Consequently, this value assignment becomes necessary to justify the unjustified line  $e_1$ . We proceed in the same way with unjustified line  $f_1$ , learning

TABLE II  
USING *demo\_recursive\_learning()*

0. learning level	1. learning level	2. learning level
(generally valid signal values)		<u>unjust. line <math>e_1 = 0</math>:</u>
$i_1 = 0$ (unjust.)	<u>unjust. line <math>i_1 = 0</math>:</u>	1. justif.: $a_1 = 0$
$j = 1$ (unjust.)		=> $a_2 = 0$
	1. justif.: $g_1 = 0$	=> $e_2 = 0$
	enter	
	=> $e_1 = 0$ (unjust.)	2. justif.: $b_1 = 0$
	learning ->	=> $b_2 = 0$
	=> $f_1 = 0$ (unjust.)	=> $e_2 = 0$
	enter next	
	recursion ->	
		$e_2 = 0$ <=====
		<u>unjust. line <math>f_1 = 0</math>:</u>
		1. justif.: $c_1 = 0$
		=> $c_2 = 0$
		=> $f_2 = 0$
		2. justif.: $d_1 = 0$
		=> $d_2 = 0$
		=> $f_2 = 0$
		<=====
	$f_2 = 0$	
	=> $g_2 = 0$	
	=> $i_2 = 0$	
	=> $k = 1$	
	2. justif.: $h_1 = 0$	
	=> $h_2 = 0$	
	=> $i_2 = 0$	
	=> $k = 1$	
$k = 1$	<=====	
$i_2 = 0$		
	<u>unjust. line <math>j = 1</math>:</u>	
	...	

that  $f_2 = 0$  is also necessary. Returning to the first learning level, the implications can be completed to yield the necessary assignments  $k = 1$  and  $i_2 = 0$ , completing the learning.

Two key points to be noted are: 1) all signal assignments that have been made during learning in each level have to be erased again as soon as learning is finished in the current level of recursion and 2) only those values that are learned to be necessary are transferred to the next lower level of recursion. Furthermore, one important aspect of this procedure is that the unjustified lines are considered separately. At each unjustified line, we try the different justifications and then move to the next unjustified line. A natural question arises as to how one takes into account that some necessary assignments result from the presence of several unjustified lines, if the justifications at one unjustified line are not tried in combination with the justifications at the other unjustified lines? Note that the necessary assignment,  $k = 1$ , in the

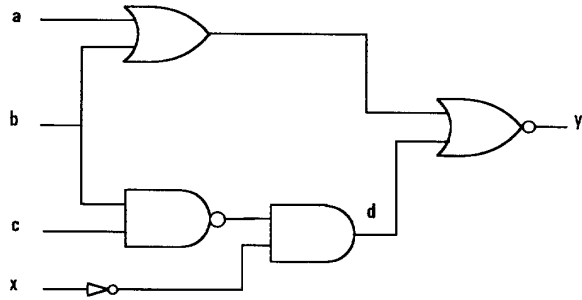


Fig. 2. Incomplete forward implications.

above example, is due to both unjustified line  $i_1 = 0$  and  $j = 1$  and is correctly derived by *demo\_recursive\_learning()*. Specifically, the interdependence of different unjustified lines is accounted for because forward implications are performed that check the consistency of the justification attempts against *all* other unjustified lines. However, the completeness of the forward implications is not always guaranteed, and therefore this preliminary version of the recursive learning routine *demo\_recursive\_learning()* may *fail* to identify all necessary assignments. To understand what extension has to be made to identify *all* necessary assignments, consider Fig. 2, which provides an example of how forward implications can be incomplete.

Consider signals  $x$  and  $d$  in Fig. 2. No forward implication can be made for signal  $d$  after the assignment  $x = 0$  has been made. However, it is easy to see that if  $x = 0$ , both assignments  $d = 0$  and  $d = 1$  result in  $y = 0$ . Hence, the forward implication  $x = 0 \Rightarrow y = 0$  is true.

In practice, incompleteness of forward implications seems a minor problem. When learning is performed for a particular unjustified line, and when necessary assignments are missed because of incomplete forward implications, then there is often some other unjustified line for which these necessary assignments can be learned.

This above incompleteness of forward implications can be illustrated using Figs. 1 and 2. If we add the circuitry of Fig. 2 between the pads  $x_e$  and  $y_e$  in Fig. 1 such that signal  $x$  of Fig. 2 is connected to  $x_e$  and  $y$  is connected to  $y_e$ , it can be observed that learning for unjustified line  $i_1$  will no longer yield the necessary assignments  $k = 1$  and  $i_2 = 0$ . However, (as the reader may verify) the necessary assignments  $k = 1$  and  $i_2 = 0$  can still be identified when learning is performed for unjustified line  $j$ . Experiments show that this is a frequent phenomenon that can be accounted for, as explained in Section V. Nevertheless, procedure *demo\_recursive\_learning()* can miss necessary assignments because of incomplete forward implications. The reader may verify, as an exercise, that learning at line  $j$  will also fail to identify  $k = 1$  and  $i_2 = 0$  if we add similar circuitry as in Fig. 2 (remove the inverter and replace NOR by OR) between the pads  $x_g, y_g$  and  $x_h, y_h$ . The reason for this incompleteness is that unjustified lines are not the only logic constraints at which learning has to be initialized. To *overcome* this problem, the concept of unjustified lines will be generalized. Consider the previous example; if recursive learning, as explained, is applied in Fig.

2 to signal  $d$ , then the forward implication  $x = 0 \Rightarrow y = 0$  will be the final result. Hence, from this we can deduce that recursive learning should be applied not only to the unjustified lines, but also to certain other signals, to make it complete. This problem is addressed in the next section by defining the concept of *unjustified gates*, on which recursive learning should be applied to make it identify *all* the necessary assignments. In the earlier example concerning Fig. 2, where  $y = 0$ , recursive learning was applied on the AND gate whose output signal is  $d$ , which is an unjustified gate. The first input for this gate is the output of a NOT gate and the second input is the output of the NAND gate, as shown. There are two justifications possible at this unjustified gate: 1) The output of the NAND and AND both taking a value of 1, and 2) The output of the NAND and AND both taking a value 0. By performing direct implications of these justifications, it can be seen that both these justifications lead to  $y = 0$ , which yields a necessary assignment.

### III. RECURSIVE LEARNING TO DETERMINE ALL NECESSARY ASSIGNMENTS

In a FAN-type algorithm, necessary assignments are derived in two different ways. The first is based on a structural examination of conditions for fault detection [3], [4]. Secondly, it is the task of an *implication procedure* to derive necessary assignments that result from previously made signal assignments. The concept of recursive learning allows the design methods to identify all necessary assignments for both cases. In Section 3.1, a technique is presented that can make *all* implications for a given situation of value assignments with absolute precision, time permitting. Section 3.2 introduces a technique to derive *all* necessary assignments resulting from the requirement to propagate the fault signal to at least one primary output.

#### A. The Precise Implication Procedure

It was pointed out in the previous section that the concept of justified lines must be generalized to guarantee the completeness of the algorithm. Here, we introduce the more general concept of *unjustified gates*. Def. 1 uses the common notation of a "specified signal," by which we understand a signal with a fixed value. In the common logic alphabet of [11],  $B_5 = (0, 1, D, \bar{D}, X)$ , a signal is specified if it has one of the values '0', '1', 'D' or ' $\bar{D}$ '. It is unspecified if it has the value 'X'.

*Def. 1:* Given a gate  $G$  that has at least one specified input or output signal: Gate  $G$  is called *unjustified* if there are one or several unspecified input or output signals of  $G$  for which it is possible to find a combination of value assignments that yields a conflict at  $G$ . Otherwise,  $G$  is called *justified*.

The concept of unjustified gates can be used to give a definition of *precise implications* and *necessary assignments*:

*Def. 2:* For a given circuit and a given situation of value assignments, let  $f$  be an arbitrary but unspecified signal in the circuit and  $V$  be some logic value. If all consistent combinations of value assignments for which no unjustified gates are left in the circuit contain the assignment  $f = V$ ,

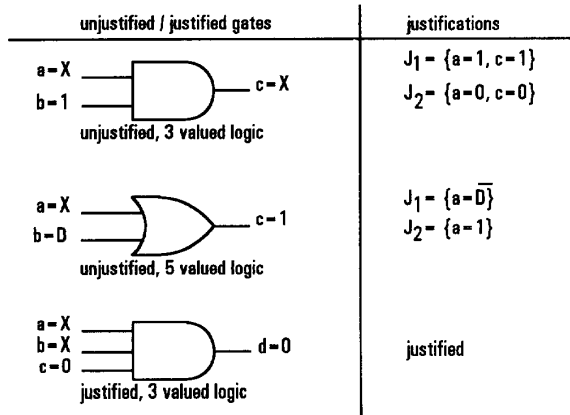


Fig. 3. Justification for unjustified gates.

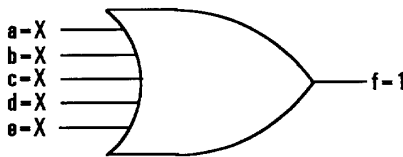


Fig. 4. Determining a complete set of justifications.

then the assignment  $f = V$  is called *necessary* for the given situation of value assignments. Implications are called *precise* or *complete* when they determine all necessary assignments for a given situation of value assignments.

To determine necessary assignments, we will consider justifications:

**Def. 3:** A set of signal assignments,  $J = \{f_1 = V_1, f_2 = V_2, \dots, f_n = V_n\}$ , where  $f_1, f_2, \dots, f_n$  are unspecified input or output signals of an unjustified gate  $G$  is called *justification* for  $G$  if the combination of value assignments in  $J$  makes  $G$  justified.

The left column of Fig. 3 shows examples of unjustified and justified gates. The right column depicts the corresponding justifications.

**Def. 4:** Let  ${}^G C$  be a set of  $m$  justifications  $J_1, J_2, \dots, J_m$  for an unjustified gate  $G$ . If there is at least one justification  $J_i \in {}^G C, i = 1, 2, \dots, m$  for any possible justification  $J^*$  of  $G$ , such that  $J_i \subseteq J^*$ , then set  ${}^G C$  is called *complete*.

For a given unjustified gate, it is straightforward to derive a complete set of justifications. In the worst case, this set consists of all consistent combinations of single assignments representing a justification of the given gate. Often, though, the set can be smaller, as shown in Fig. 4.

The following represents a complete set of justifications:  $C = \{J_1, J_2, J_3, J_4, J_5\}$  with  $J_1 = \{a = 1\}$ ,  $J_2 = \{b = 1\}$ ,  $J_3 = \{c = 1\}$ ,  $J_4 = \{d = 1\}$ ,  $J_5 = \{e = 1\}$ . Note that for example the justification  $J^* = \{a = 1, b = 0\}$  does not have to be in  $C$  since all assignments in  $J_1$  are contained in  $J^*$ .

The concept of justifications for unjustified gates is essential toward understanding how learning is used to derive necessary assignments. Assignments are obviously necessary for the justification of a gate if they have to be made for *all* possible

justifications. By definition, all assignments that have to be made for all justifications that represent a complete set of justification, also have to be made for any order justification at the respective gate. Hence, for a given gate, it is sufficient to consider a complete set of justifications in order to learn assignments that are necessary for all justifications.

All learning operations rely on a basic implication technique. As in [12], we shall call these implications *direct implications*:

**Def. 5:** *Direct implications* are implications that can be performed by only evaluating the truth table for a given gate with a given combination of value assignments at its input and output signals, and by propagating the signal values according to the connectivity in the circuit.

A well-known example of direct implications in combinational circuits are the implications performed in FAN [3].

**Notation:**  $r$ : integer number for the depth of recursion  
 ${}^0 U = \{G_1, G_2, G_3, \dots\}$  is the set of all unjustified gates as they result from the current state of the test generation algorithm.  
 ${}^{G_x} J^r = \{f_1 = V_1, f_2 = V_2, \dots\}$  is a set of assignments that represents a justification for some gate  $G_x$  in a given recursion level  $r$ .

${}^{G_x} C^r = \{J_1, J_2, J_3, \dots\}$  is a complete set of justifications for a given gate  $G_x$  in a given recursion level  $r$ .

${}^{J_x} U^r = \{G_1, G_2, G_3, \dots\}$  is a set of unjustified gates in recursion level  $r$  as it results from a given justification  $J_x$ .

$r_{\max}$ : maximum recursion depth

Table III depicts procedure *make\_all\_implication()*, which is able to make precise implications for a given set of unjustified gates. Note that the list of unjustified gates being set up in every level of recursion contains all new unjustified gates, but must also include unjustified gates of a previous recursion level if these gates have had an event in the current level of recursion.

**Theorem 1:** *The implication procedure in Table III makes precise implications; i.e., a finite  $r_{\max}$  always exists such that  $\text{make\_all\_implication}(0, r_{\max})$  determines all necessary assignments for a given set of unjustified gates,  ${}^0 U$ .*

**Proof:** *Preliminary Remarks* Making precise implications means identifying all signal values that are uniquely determined due to the unjustified gates contained in the set  ${}^0 U$ . Let  $V$  be a logic value and let us assume for some signal  $f$  that the assignment  $f = V$  is necessary for the justification of a gate  $G_x$  in an arbitrary recursion level  $r$ . What this means is that one of the following two cases must be fulfilled for each justification  ${}^{G_x} J_i^r \in {}^{G_x} C^r$ :

**Case 1:** The direct implications for the set of assignments  $J_i$  at  $G_x$  yield  $f = V$ .

**Case 2:** The assignment  $f = V$  is necessary for the justification of at least one gate in the set of justified gates  ${}^{J_i} U^{r+1}$ .

In the first case, the necessary assignment is recognized and learned when learning in level  $r + 1$ . In the latter, deeper recursion level are entered.

**Complete Induction:**

1) Take  $r = r_{\max} - 1$ :

The more recursions performed, the more assignments are made; i.e., for all unjustified gates in  ${}^0 U$ , the recur-

TABLE III  
PRECISE IMPLICATION PROCEDURE

```

initially: r:=0;
make_all_implications(r, r_max)
{
  make all direct implications and set up a list Ur of
  resulting unjustified gates
  if r < r_max : learning
  {
    for each gate Gx, x=1,2,..., in Ur: justifications
    {
      set up list of justifications GxCr
      for each justification Ji ∈ GxCr:
      {
        - make the assignments contained in Ji
        - make_all_implications(r+1, r_max )
      }

      if there is one or several signals f in the circuit, which
      assume the same logic value V for all consistent
      justifications Ji ∈ GxCr then learn: f=V is uniquely
      determined in level r, make direct implications for all
      learned values in level r

      if all justifications are inconsistent, then learn: given
      situation of value assignments in level r is inconsistent
    }
  }
}
  
```

sive call of *make\_all\_implications()*, will always reach a level *max*, such that  $J_i U^{\max} = \emptyset$  for all justifications  $J_i \in G_x C^{\max-1}$  and for all gates  $G_x$  in  $U^{\max-1}$ . This is the case when the implications have reached the primary inputs or outputs. If there is a necessary assignment  $f = V$  in level  $r = \max$ , we will always recognize it, since  $U^{\max} = \emptyset$  and for any necessary assignment, Case 1 must be fulfilled. This means that *all* necessary assignments have been learned for all gates in all  $U^{\max-1}$  that result from arbitrary  $J_i^{\max-2} \in C^{\max-2}$ , which belong to an arbitrary  $G_x \in U^{\max-2}$

- 2) Assume that we know all necessary assignments for all unjustified gates in all sets  $U^n$  for arbitrary  $J_i^{n-1} \in C^{n-1}$  for arbitrary  $G_x \in U^{n-1}$ .
- 3) Then, since it is guaranteed with the above assumption that all uniquely determined values be known that can be implied for all justifications in  $G_x C^{n-1}$ , the procedure *make\_all\_implication()* will correctly identify all necessary assignments for the corresponding gate  $G_x \in J U^{n-1}$ . (These are the signal values common for all justification  $G_x C^{n-1}$ .) The above is true for any gate  $G_x \in J_i U^{n-1}$ , where  $J_i$  is some justification  $J_i^{n-2}$  for some gate in  $U^{n-2}$ . Hence, all necessary

TABLE IV  
DEMONSTRATING PROCEDURE MAKE ALL IMPLICATIONS()

0. learning level (generally valid signal values)	1. learning level	2. learning level
$p = 1$ (unjust.)	<b>for unjust. gate G6 :</b>	<b>for unjust. gate G1:</b>
enter learning ->	1. justif.: $q = 0, r = 0$ => $k = 0$ (G1 unjust.) => $i = 0$ (G2 unjust.) => $m = 0$ (G3 unjust.) => $n = 0$ (G4 unjust.)	1. justification $c = 0$ => $e = 1$ => $f = 0$ (since $l = 0$ ) => $j = 0$ (since $n = 0$ ) => inconsistency at b
	enter next recursion ->	2. justification $d = 0$ => $g = 1$ => $h = 0$ (since $m = 0$ ) => $j = 1$ => $i = 0$ (since $n = 0$ ) => inconsistency at a
	1. justification inconsistent	current situation of value assignments inconsistent <=====
	2. justif.: $q = 1, r = 1$ $r = 1$ : G5 unjust.	<b>for unjust. gate G5:</b>
	enter next recursion ->	1. justification $k = 1$ => ...
	$q = 1$ and $r = 1$ are common for all consistent justifications (there is only one)	2. justification $l = 1$ => ...
$q = 1$ $r = 1$	<=====	3. justification $m = 1$ => ...
		4. justification $n = 1$ => ... (no new information learned) <=====

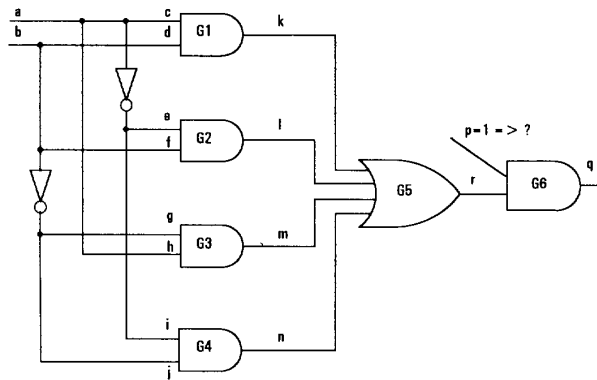


Fig. 5. Making precise implications for  $p = 1$ .

assignments are recognized for all unjustified gates in all sets  $U^{n-1}$  for arbitrary  $J_i^{n-2} \in C^{n-2}$  for arbitrary  $G_x \in U^{n-2}$ . By complete induction we conclude that we learn all necessary assignments for all unjustified gates  $G_x$  in  $U$ .

Fig. 5 shows some combinational circuitry to illustrate *make\_all\_implication()*. Table IV lists the single steps to per-

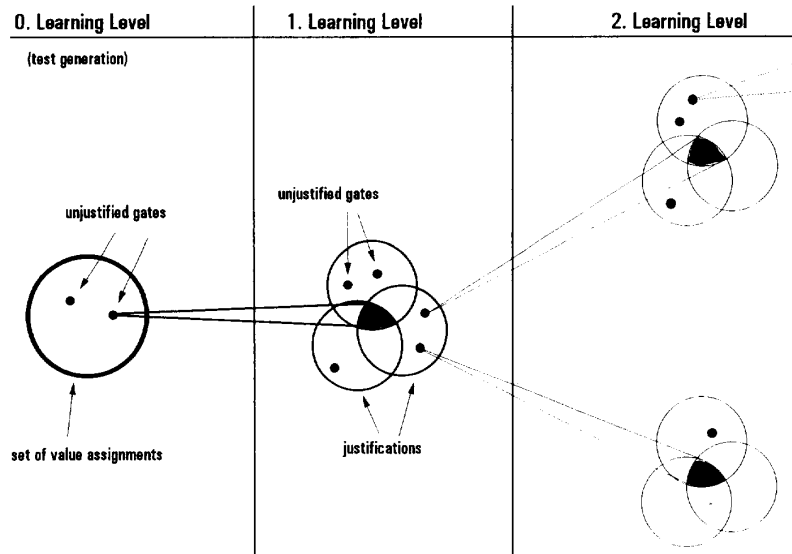


Fig. 6. Schematic illustration of recursive learning.

form the implication  $p = 1 \Rightarrow q = 1$ . Note that the learning technique [6], [7], [11] cannot perform this implication.

Fig. 6 depicts a scheme useful to better understanding the general procedure of recursive learning and the proof of Theorem 1.

During the test generation process, optional assignments are made. After each optional assignment, the resulting necessary assignments must be determined. This is the task of the implication procedure. Many necessary assignments can be determined by performing direct implications only. Direct implications can handle the special case where there is only one possible justification for an unjustified gate. (Note that this represents another possibility to define "direct" implications.) The left column in Fig. 6 shows the situation as it occurs after each optional assignment during the test generation process. After performing direct implications, we have obtained a situation of value assignments where only those unjustified gates (dark spots in Fig. 6) are left that allow for more than one justification. These are examined by learning. Recursive learning examines the different justifications for each unjustified gate, which results in new situations of value assignments in the first learning level. If value assignments are valid for all possible justifications of an unjustified gate in level 0, i.e., if they lie in the intersection of the respective sets of value assignments in learning level 1 (shaded area), then they actually belong to the set of value assignments in level 0. This is indicated schematically in Fig. 6. However, the set of value assignments in learning level 1 may be incomplete as well because they also contain unjustified gates and the justifications in level 2 have to be examined. This is continued until the maximum recursion depth  $r_{\max}$  is reached.

This immediately leads to the question: how deep do we have to go in order to perform precise implications? Unfortunately, it is neither possible to predict how many levels of recursion are needed to derive all necessary assignments, nor is it possible to determine if all necessary assignments

have been identified after learning, with a certain recursion depth, has been completed. The choice of  $r_{\max}$  is subject to heuristics and depends on the application for which recursive learning is used. For test generation, an algorithm to choose  $r_{\max}$  will be presented in Section 4.1.

In general, it can be expected that the maximum depth of recursion to determine all necessary assignments is relatively low. This can be explained as follows: Note, that value assignments in level  $i + 1$  are only necessary for level  $i$  if they lie within the intersection in level  $i + 1$ ; to be necessary in level  $i - 1$  they also have to be in the intersection of level  $i$ , and so forth. It is important to realize, however, that we are only interested in the necessary assignments of level 0. It is not very likely that a value necessary in level 10 also lies in the corresponding intersections of level 9, 8, 7, ..., 1 and, hence, is not likely to be necessary in level 0. Necessary assignments of level 0 are usually determined by only considering a few levels of recursion. This corresponds to the plausible concept that unknown logic constraints (necessary assignments) must lie in the "logic neighborhood" of the known logic constraints from which they are caused.

Intuitively, a lot of recursions are only needed if there is a lot of redundant circuitry. Look at the circuits in Figs. 1, 2, and 5: Necessary assignments are only missed by direct implications because the shown circuits contain sub-optimal circuitry. In the scheme of Fig. 6, the intersection of justifications (shown as shaded areas) indicate logic redundancies in the circuit. In fact, making precise implications and identifying sub-optimal circuitry seem to be closely related. This promises that recursive learning is also useful in logic optimization.

Use Fig. 6 to understand the proof of Theorem 1: It is important to realize that the process of recursive learning terminates, even if the parameter  $r_{\max}$  in *make\_all\_implication*( $r, r_{\max}$ ), is chosen to be infinite. At some point, the justifications must reach the primary inputs and outputs so that no new unjustified gates requiring further recursions can be caused.

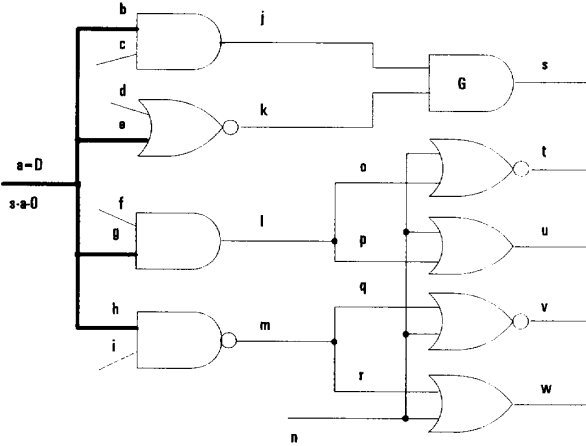


Fig. 7. Necessary assignments for fault propagation.

In Fig. 6, such justifications are represented by circles that do not contain dark spots. If the individual justifications for a considered unjustified gate do not contain unjustified gates, it is impossible (because of Def. 1) that these sets of value assignments produce a conflict with justifications of some other unjustified gates. Since a complete set of justifications is examined and the same argument applies to every unjustified gate in the previous recursion level, it is guaranteed that *all* necessary assignments for the previous recursion level are identified. This is used in Step 1 of the complete induction for Theorem 1. If all necessary assignments are known in a given recursion level, the intersections of the complete sets of justifications yield all necessary assignments for the previous recursion level and steps 2 and 3 of the complete induction are straightforward.

### B. Determining All Necessary Assignments for Fault Propagation

In principle, the problem of test generation is solved with a precise implication technique as given in Section 3.1. Observability constraints can always be expressed in terms of unjustified lines by means of Boolean difference. However, most atpg-algorithms use the concept of a “*D*-frontier” [2]. This makes it easier to consider topological properties of the circuit [4]. In this section, we present a technique to identify all necessary assignments that are due to the requirement of propagating the fault signal to at least one primary output. Analogous to the previous section where we injected justifications at unjustified gates in order to perform precise implications, this section shows how recursive learning can derive all conditions for fault propagation by injecting *sensitizations* [3] at the *D*-frontier.

The *D*-frontier in a recursion level  $r$  shall be denoted  $F^r$  and consists of all signals that have a faulty value and a successor signal that is still unspecified. Fig. 7 shows an example for a *D*-frontier. If we set up the fault signal  $D$  for the stuck-at-0 fault at a line  $a$ , we obtain  $F^0 = \{b, e, g, h\}$ .

Table V lists procedure *fault\_propagation\_learning()*. In Table VI, how *fault\_propagation\_learning* is used to determine

TABLE V  
PROCEDURE FAULT PROPAGATION LEARNING()

```

fault_propagation_learning(r, rmax)
{
  for all signals  $f_D \in F^r$  : sensitization
  {
    successor_signal =  $f_D$ ;
    while ( successor_signal has exactly one successor
           (no fanout stem))
    {
      fault_value := value of successor_signal;
      successor_signal := successor of successor_signal
      if (successor_signal is output of inverting gate )
        assign: value of successor_signal := INV(fault_value)
      else
        assign: value of successor_signal := fault_value
    }
    make_all_implications(r+1, rmax);
    set up list of new D- frontier  $F^{r+1}$ ;
    if ( $r < r_{max}$  and current sensitization is consistent )
      fault_propagation_learning(r+1, rmax);
  }
  if there is one or several signals  $f$  in the circuit, which each
  assume the same logic value  $V$  for all non-conflicting
  sensitizations, then learn:  $f=V$  is uniquely determined in
  level  $r$ , make direct implications for learned values in level  $r$ 

  if all sensitizations result in a conflict, then learn:
  fault propagation in level  $r$  impossible (conflict)
}

```

the necessary assignment  $n = 0$  in Fig. 7 is explained step by step.

Procedure *fault\_propagation\_learning()*, which calls procedure *make\_all\_implications()*, learns all assignments that are necessary to sensitize *at least* one path from the fault location to an arbitrary output. Note that we are *not* only considering single path sensitization. Along every path that is sensitized in procedure *fault\_propagation\_learning()*, gates becomes unjustified if there is more than one possibility to sensitize them. This is demonstrated in Table VI for gate  $G$  in Fig. 7.

Procedure *fault\_propagation\_learning()* as given in Table V does not show how to handle XOR-gates. However, the extension to XOR-gates is straightforward. XOR-gates, like XNOR-gates, always allow for more than one way to sensitize them. Therefore, the fault propagation has to stop there and the different possibilities to propagate the fault signal have to be tried after the usual scheme for unjustified gates.

## IV. TEST GENERATION WITH RECURSIVE LEARNING

### A. An Algorithm to Choose the Maximum Recursion Depth $r_{max}$

There are many possibilities to design a test generation algorithm with recursive learning. There is unlimited freedom

TABLE VI  
DEMONSTRATING FAULT PROPAGATION LEARNING()

0. learning level	1. learning level	2. learning level
(generally valid signal values) $F^0 = \{b, e, g, h\}$ enter learning ->	<u>D- frontier signal b:</u> 1. sensitization: successor of b: $\Rightarrow j = D, c = 1$ successor of j: $\Rightarrow s = D, (unjust.)$ enter next recursion ->  1. sensitization failed <=====	for unjust. gate G: 1. justification $k = 1$ $\Rightarrow$ inconsistent with $e = D$ 2. justification $k = D$ $\Rightarrow$ inconsistent with $e = D$
	<u>D- frontier signal e:</u> 2. sensitization: successor of e: $\Rightarrow k = \bar{0}, d = 0$ successor of k: $\Rightarrow s = \bar{0} (unjust.)$ enter next recursion ->  2. sensitization failed <=====	for unjust. gate G: 1. justification $j = 1$ $\Rightarrow$ inconsistent with $b = D$ 2. justification $j = \bar{0}$ $\Rightarrow$ inconsistent with $b = D$
	<u>D- frontier signal g:</u> 3. sensitization: successor of g: $\Rightarrow l = D, f = 1$ several successors of l: $\Rightarrow F^1 = \{o, p\}$ enter next recursion ->  $n = 0$	<u>D- frontier signal o:</u> 1. sensitization: successor of o: $\Rightarrow t = \bar{0}, n = 0$  <u>D- frontier signal p:</u> 2. sensitization: successor of p: $\Rightarrow u = D, n = 0$ <=====
	<u>D- frontier signal h:</u> 4. sensitization: successor of h: $\Rightarrow m = \bar{0}, i = 1$ several successors of m: $\Rightarrow F^1 = \{q, r\}$ enter next recursion ->  $n = 0$	<u>D- frontier signal q:</u> 1. sensitization: successor of q: $\Rightarrow v = D, n = 0$  <u>D- frontier signal r:</u> 2. sensitization: successor of r: $\Rightarrow w = \bar{0}, n = 0$ <=====

to make optional assignments. We are not bound to the strict scheme of the decision tree in order to guarantee the completeness of the algorithm; it is possible to "jump around" in the search space. Note that this allows attractive possibilities for new heuristics. To guarantee completeness, we only have to make sure that the maximum recursion depth is eventually incremented. Of course, it is wise to keep the maximum recursion depth  $r_{max}$  as small as possible so as not to spend much on learning operations. Only if the precision is not sufficient to avoid wrong decisions is it sensible to increment  $r_{max}$ .

There are many possibilities for choosing  $r_{max}$ . To examine the performance of recursive learning, we combined it with the FAN-algorithm and used the following strategy to generate test vectors: the algorithm proceeds like in FAN and makes optional assignments in the usual way. In the same way as for the decision tree, all optional assignments are stored in a stack. Whenever, a conflict is encountered, we proceed as shown in Fig. 8. By a conflict, we mean that the previous

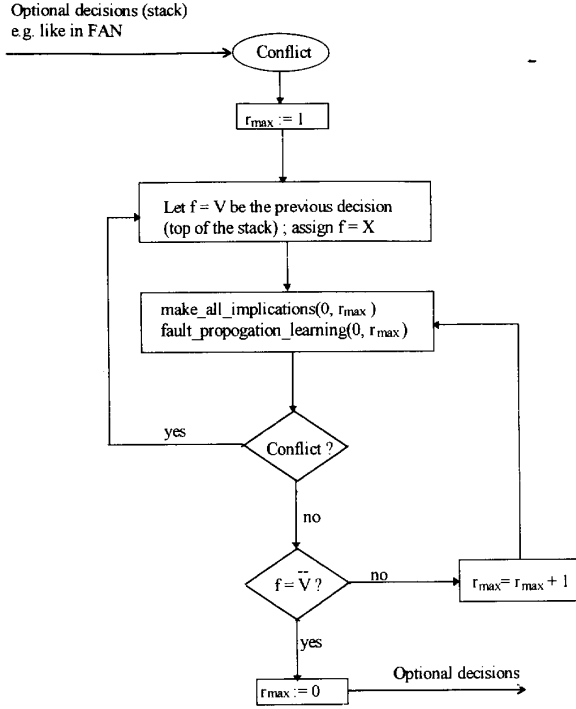


Fig. 8. Algorithm for choosing  $r_{max}$ .

decisions have either led to an inconsistent situation of value assignments or that there is no more possible propagation path for the fault signal ( $X$ -path-check failed). The idea behind the routine in Fig. 8 is that we use learning only to leave the nonsolution areas as quickly as possible. After a conflict has occurred, the previous decision is erased; i.e., the signal at the top of the stack is removed and its value is assigned to 'X'. Now, the resulting situation of value assignments is examined with increased recursion depth. If this leads to a new conflict, another decision has to be erased. If there is no conflict, this can mean two things: either the current precision  $r_{max}$  is not sufficient to detect that there is still a conflict or we have actually returned into the solution area of the search space. Therefore, it is checked if the opposite of the previous (wrong) assignment is one of the assignments that have been learned to be necessary. This is a good heuristic criterion to determine whether the precision has to be. This criterion also makes sure that we can never enter the same nonsolution area twice, and the algorithm in Fig. 8 guarantees the completeness of test generation and redundancy identification without the use of a decision tree.

Note that the procedure in Fig. 8 is only one out of many possibilities to integrate recursive learning into existing test generation tools. This algorithm has been chosen because it allows a fair comparison of recursive learning with the decision tree. With the algorithm of Fig. 8, we initially enter exactly the same nonsolution area of the search space as with the original FAN-algorithm. The comparison is how fast the nonsolution areas are left either by conventional backtracking or by recursive learning.

One disadvantage of the above procedure is that in some cases of redundant faults the algorithm initially traverses very deep into nonsolution areas and recursive learning has to be repeated many times until all optional value assignments (those will be all wrong) are erased step by step. Our current implementation therefore makes use of the following intermediate step (not shown in Fig. 8 for reasons of clarity): when the algorithm of Fig. 8 reaches the point where the maximum depth of recursion has to be incremented, we perform recursive learning with increment recursion depth first only to the situation of values assignments that results if all optional value assignments are removed. If a conflict is encountered, the fault is redundant and we are finished. Otherwise, we proceed, as shown in Fig. 8, i.e., we perform recursive learning, with the optional value assignments as given on the stack.

### B. Compatibility and Generality of the Approach

Many heuristics have been reported in the past to guide decision-making during test generation. Most of these techniques are equally suitable in combination with recursive learning. Just as they reduce the number of backtracks in the decision tree, they similarly reduce the number of recursions needed when recursive learning is performed. In particular, it seems wise to consider the static learning technique of [6], [7] to pre-store indirect implications. Furthermore, a method to identify equivalent search states [13] and dominance search states [14] can also be applied to recursive learning. This is because the formulation of a search state as *E-Frontier* [13] can also be applied to the different sets of value assignments that occur during recursive learning.

Note that recursive learning in this paper has been based on the common procedure to perform direct implications. It is also possible to use other implication procedures as the basic "workhorse" of recursive learning. Finally, it is possible to use recursive learning and the decision tree at the same time, to combine the advantage of both searching methods.

In this work, we have examined the performance of recursive learning only for combinational circuits. However, the approach is also feasible for sequential circuits. Even for hierarchical approaches, recursive learning can be used as an alternative to the decision tree. Although all considerations in this paper have been based on the gate level, it is straightforward to extend the concept of justifications for unjustified gates to high level primitives. A large logic block that is unjustified may have a lot of justifications. However note that we only have to continue to try justifications for a given unjustified gate (or high level primitive), as long as there is at least one common value for *all* consistent previous justifications. Therefore, many justifications have to be tried only if there are actually common values for many justifications.

### C. The Intuition Behind Recursive Learning

What is the intuition behind this approach and why is it faster than the traditional searching method? At first glance, recursive learning may seem similar to the search based on the decision tree as applied in the *D*-algorithm [11]. Note that also the decision tree can be used to derive all necessary

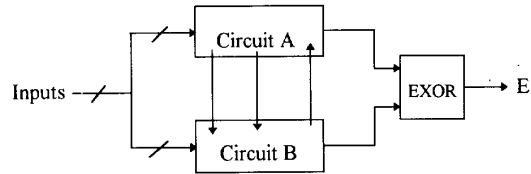


Fig. 9. Logic verification based on indirect implication.

assignments: for a given set of unjustified gates the justifications can be tried by making optional value assignments, which are added to a decision tree. We exhaust the decision tree; i.e., we continue even after a sufficient solution has been found and obtain all necessary assignments by checking what assignments are common to all sufficient solutions. Like limiting the depth of recursion for recursive learning, we can limit the number of optional decisions that we put into the decision tree so that we only examine the neighborhood of the given unjustified gates.

However, there is a fundamental difference between this approach and recursive learning. As pointed out in Section II, recursive learning examines the different unjustified gates separately, one after the other, whereas the decision tree (implicitly) enumerates all combinations of justifications at one gate with all combinations of justifications at the other gates. This results in an important difference between the two methods; recursive learning *only* determines all *necessary* assignments; in contrast to the decision tree, it neither has to derive all sufficient solutions explicitly nor implicitly to obtain all necessary assignments. Consequently, the behavior of recursive learning is quite different from a decision tree-based search. In recursive learning, signal values are only injected temporarily. Only value assignments that are necessary in the current level of recursion are retained. With the decision tree, however, all assignments that are not proven to be wrong are maintained. If a wrong decision has occurred, it can happen that this decision is "hidden" behind a long sequence of subsequent good decisions, so that the conflict occurs only many steps later. At this point, much enumeration is necessary with the decision tree until the wrong decision is finally reversed. For the precise implication, however, a lot of searching is needed if necessary assignments are "hidden" by large redundant circuitry. Roughly, it is possible to state that the computational costs to perform precise implications depend on the size of the redundant circuit structures. The relationship between redundancy in the circuit and the complexity of performing precise implications, at this point, is only understood at an intuitive level and is subject to future research.

## V. EXPERIMENTAL RESULTS

To examine the performance of recursive learning, we use the FAN-algorithm. For comparison, we use both the original FAN-algorithm with the decision tree as well as a modified version where we replaced the decision tree by recursive learning. *No additional techniques were used.* Recursive learning is performed, as was shown in Section 4.1.

TABLES VII  
EXPERIMENTAL RESULTS FOR REDUNDANT FAULTS (SUN SPARC 10/51)

Results if only redundant faults are targeted		FAN with DECISION TREE (bt. limit of 1000)			FAN with RECURSIVE LEARNING						
circuit	no. faults targeted	no. of backtracks	time [s]	ab.	time [s]	ab.	learning levels:				
							r0	r1	r2	r3	r4
c432	4	3000	12	3	0.2	0	1	3	-	-	-
c499	8	0	0.1	0	0.1	0	8	-	-	-	-
c880	0	-	-	-	-	-	-	-	-	-	-
c1355	8	0	0.1	0	0.1	0	8	-	-	-	-
c1908	9	226	5	0	0.2	0	7	-	-	-	-
c2670	117	18862	207	15	112	0	81	25	0	4	7
c3540	137	5000	339	5	2	0	132	5	-	-	-
c5315	59	0	0.9	0	0.9	0	59	-	-	-	-
c6288	34	0	2	0	2	0	34	-	-	-	-
c7552	131	64733	3859	64	36	0	65	66	-	-	-
s5378	39	0	1	0	1	0	39	-	-	-	-
s9234	443	55396	4238	36	75	0	305	106	32	-	-
s13207	149	7159	1118	1	23	0	131	17	1	-	-
s15850	384	2459	367	2	31	0	360	24	-	-	-
s35932	3728	0	837	0	837	0	3728	-	-	-	-
s38417	161	4056	1207	4	47	0	153	8	-	-	-
s38584	1345	8137	3277	2	227	0	1300	37	8	-	-

There are two general aspects of recursive learning in a FAN-based environment that we use for better efficiency: first, as discussed in Section II, there are very few cases in practical life where it is necessary to perform learning at unjustified gates with an unspecified output signal. Therefore, learning for unjustified gates with an unspecified output signal shall be done with a maximum recursion level of  $r_{\max} = 5$  if the current maximum recursion level  $r_{\max}$  is larger than 5. Otherwise, no learning is performed for such gates. '5' was chosen intuitively to suppress the unnecessary recursions so that they contribute little to the total CPU-time but still guarantee the completeness of the algorithm. Second, in a FAN-based algorithm, there are two possibilities how the decision making can fail: inconsistency and  $X$ -path check failure. In the first case, we initially perform only procedure *make\_all\_implications()*. Only when the maximum recursion level exceeds 3 do we also perform *fault\_propagation\_learning()*.

To illustrate the different nature of the two searching schemes, we first compare recursive learning to the traditional

search by looking only a redundant faults. In our first experiment, we target only all redundant faults in the ISCAS85 [19] benchmarks and the seven largest ISCAS89 [20] benchmarks. The results are given in Table VII. The first column lists the circuit that is examined; the second column shows the number of redundant faults of the respective circuit. Only these are targeted by the test generation algorithm. First, we run FAN with a backtrack limit of 1000; i.e., the traditional searching scheme is used and the fault is aborted after 1000 backtracks. The third column shows the number of backtracks for each circuit. The next two columns show the CPU-time in seconds and the number of aborted faults. In the second run, we use recursive learning instead of the decision tree. Columns 6 and 7 give the CPU-time and the number of aborted faults for recursive learning. The next four columns show for how many faults the highest depth of recursion was chosen by the algorithm in Fig. 8. For example, for circuit c432, one redundancy could be identified without any learning. Three redundancies were identified in the first learning level.

Impressively, the results show the superiority of recursive learning for redundancy identification compared to the decision tree. Look, for example, at a circuit c3540: With the decision tree 5, faults are aborted after performing 1000 backtracks each. There are a total of 5000 backtracks for this circuit. Obviously, for 132 redundancies, there have been no backtracks at all. We observe an "all-or-nothing-effect" typical for redundancy identification with the decision tree. If the implications fail to reveal the conflict, the search space has to be exhausted to prove that no solution exists. This is usually intractable.

Recursive learning and the search based on the decision tree are in a complementary relationship: the latter is the search for a sufficient solution, its pathological cases being where no solution exists (redundant faults); the former is the search for all necessary conditions. If recursive learning was used to prove that a fault is testable, without constructively generating a test vector, the pathological cases are the cases where no conflict occurs; i.e., we have to exhaust the search space if a solution exists (testable faults).

Although recursive learning is always used in combination with making optional decisions to generate a test vector such as given in Fig. 8, it is not wise to use it in cases where many solutions exist that are easy to find. For those cases, it is faster to perform a few backtracks with the decision tree, as we have already shown by the results in [18]. There are many efficient ways to handle these "easy" faults. In this paper, we choose to perform 20 backtracks with the decision tree. These are split into two groups of ten backtracks, each. For the first ten backtracks, we use the FAN-algorithm with its usual heuristics. When ten backtracks have been performed, the fault is aborted and re-targeted again. The second time we use *orthogonal heuristics*. This means that we always assign the opposite value at the fanout objectives [3] of what FAN's multiple backtrace procedure suggests. This allows us to explore the search space in an orthogonal direction compared to our first attempt. For testable faults, this procedure has been shown to be very effective. As an example, in circuit c6288 when test generation was performed for all faults, 225 faults remained undetected after the first ten backtracks. After the following ten backtracks with orthogonal heuristics, only ten faults were left. Faults that remain undetected after these 20 backtracks are aborted in phase 1; they represent the difficult cases for most FAN-based atpg-tools. These pathological faults are the interesting cases when comparing the performance of the two searching techniques.

Table VIII shows the results of test generation of the ISCAS85 benchmarks and for the 7 largest ISCAS89 benchmarks. After each generated test vector, fault simulation is used to reduce the fault list (faultdropping). No random vectors are used. The first two columns list the circuits under consideration and the number of faults that have been targeted. Columns 3 to 5 show the results of the first phase, in which FAN is performed using its original and orthogonal heuristics with a backtrack limit of ten for each pass. Column 3 gives the number of faults that are identified as redundant, and column 4 lists the CPU-times in seconds. Column 5 gives the figures for the aborted faults. All faults aborted in phase 1 are re-targeted

in the second phase, in which we compare the performance of recursive learning to the search based on the decision tree. The meaning of columns 6 to 15 is analogous to Table VII.

The results in Tables VIII and IX clearly show the superiority of recursive learning to the traditional searching method in test generation. There are no more aborted faults and the CPU-times for the difficult faults are very short when recursive learning is used. A closer study of the above tables shows that the average CPU-times for each difficult fault is nearly in the same order of magnitude as for the easy faults (with only few exceptions). The results show that recursive learning can replace the "whole bag of tricks" that has to be added to the FAN algorithm if full fault coverage is desired for the ISCAS benchmarks. The implementation of our base algorithm is rather rudimentary, so that a lot of speedup can still be gained by a more sophisticated implementation. Since the focus of this paper is the examination of a new searching method and not presentation of a new tool, no effort has been made to combine recursive learning with a selection of state-of-the-art heuristics as they are used for example in [17].

Recursive learning does not affect the speed and memory requirements of atpg-algorithms as long as it is not invoked; there is no preprocessing or pre-storing of data. This is an important aspect if test generation is performed under limited resources, as pointed out [21]. If recursive learning is actually invoked, some additional memory is necessary in order to store the current situation of value assignments in each recursion level. The different flags that steer the implication procedure and store the current signal values at each gate have to be handled separately in each level of recursion. As a rough estimate, this results in an overhead of 25 Bytes for each gate in the circuit if we assume that there are 5 flags to steer the implications and a maximum recursion depth of 5. For a circuit with 100,000 gates we obtain an overhead of 2.5 Mbytes, which is usually negligible.

## VI. RECURSIVE LEARNING IN LOGIC VERIFICATION AND OPTIMIZATION

Recent years have seen much progress in the automation of the design process for large integrated circuits. Tools for automatic synthesis play a crucial role in the progress of the VLSI industry. Still in many cases, the designer must resort to manual modifications, or utilization of certain tailor-made custom software and algorithms to fulfill the special requirements of a particular design. Because the size of the circuits grows continuously in this phase of the design process, errors are highly likely, the human designer possessing little insight into the functionality of the design—especially after automatic synthesis procedures have been applied in an earlier design phase. Verification techniques have, therefore, become extremely important, with considerable research currently being conducted toward development of efficient verification methods for the various steps in the design process.

Recursive learning-based techniques have been shown to be very useful [22], [23] in (formal) logic verification problems for combinational circuits; i.e., the problem of identifying whether two circuits are functionally equivalent. Two com-

TABLE VIII  
EXPERIMENTAL RESULTS FOR TEST GENERATION WITH FAULT DROPPING (SUN SPARC 10/51)

Results for collapsed faultlist with faultdropping		1. PHASE (eliminate easy faults)			2. PHASE for DIFFICULT FAULTS (aborted in 1. phase)									
		FAN with backtrack limit of 10+10			FAN with DECISION TREE (bt. limit of 1000)			FAN with RECURSIVE LEARNING						
circuit	no. faults targeted	red.	time [s]	aborted	red.	time [s]	ab.	red.	time [s]	ab.	learning levels:			
											r1	r2	r3	r4
c432	93	1	1	3	0	12	5	3	0.2	0	3	-	-	-
c499	122	8	4	0	-	-	-	-	-	-	-	-	-	-
c880	95	0	2	0	-	-	-	-	-	-	-	-	-	-
c1355	185	8	12	0	-	-	-	-	-	-	-	-	-	-
c1908	178	7	11	2	2	5	0	2	0.1	0	2	-	-	-
c2670	343	98	26	19	8	243	11	19	90	0	8	0	4	7
c3540	392	127	47	5	0	276	5	5	2	0	5	-	-	-
c5315	460	59	37	0	-	-	-	-	-	-	-	-	-	-
c6288	80	34	15	0	-	-	-	-	-	-	-	-	-	-
c7552	533	67	210	64	0	3676	64	64	24	0	64	-	-	-
s5378	460	7	34	0	-	-	-	-	-	-	-	-	-	-
s9234	1230	389	267	56	19	4449	27	54	73	0	22	28	0	6
s13207	1096	133	309	16	15	1070	1	16	16	0	16	-	-	-
s15850	1295	374	803	10	8	331	2	10	7	0	10	-	-	-
s35932	4794	3728	1308	0	-	-	-	-	-	-	-	-	-	-
s38417	4021	153	1108	8	4	1074	4	8	15	0	8	-	-	-
s38584	3301	1321	890	24	22	3125	2	24	87	0	16	8	-	-

binational circuits are functionally equivalent if they respond to an arbitrary input pattern with the same output pattern.

Table IX shows the results if all faults are targeted. There is neither a random phase nor fault dropping.

A logic verification tool, HANNIBAL [22], has been developed recently. HANNIBAL proceeds to verify the functional equivalence of two combinational circuits *A* and *B* in the following manner. Consider Fig. 9; for simplicity, assume that two circuits have only one output. By combining the two circuits to form a new circuit with output *E* that is an EXOR of *A* and *B*, the logic verification problem obviously is reduced to solving the Boolean satisfiability problem for the output signal *E*. The implication procedure described here, in principle, represents a simple method to check the Boolean satisfiability of *E*: if the precise implications for  $E = 1$  produce a conflict, then it follows that  $E = 0$  and the two circuits are equivalent. If no conflict occurs,

the precise implication procedure determines all the value assignments necessary to generate a distinguishing vector. As pointed out, memory requirements grow linearly and CPU-time grows exponentially with the maximum depth of recursion. To complete the verification task in acceptable CPU-time, it becomes essential to keep the recursive depth as small as possible.

Recursive learning can often make use of "similarities" that exist between the two circuits under consideration. These similarities can, logically, be expressed as (usually indirect) implications between signals of different circuits. Recursive learning is a powerful technique toward identifying precisely these implications, when they exist. Note that these implications immediately indicate functional equivalence of internal nodes (as a special case). In [22], we have presented a prototype verification tool based on these concepts. Verification is conducted in two phases. During the first phase, we pass

TABLE IX  
EXPERIMENTAL RESULTS FOR TEST GENERATION WITHOUT FAULT DROPPING (SUN SPARC 10/51)

Results for collapsed faultlist no faultdropping, all faults are targeted		1. PHASE (eliminate easy faults)			2. PHASE for DIFFICULT FAULTS (aborted in 1. phase)									
		FAN with backtrack limit of 10+10			FAN with DECISION TREE (bt. limit of 1000)			FAN with RECURSIVE LEARNING						
		circuit	no. faults targeted	red.	time [s]	aborted	red.	time [s]	ab.	red.	time [s]	ab.	learning levels:	
r1	r2												r3	r4
c432	524	1	7	3	0	12	3	3	0.2	0	3	-	-	-
c499	758	8	19	0	-	-	-	-	-	-	-	-	-	-
c880	942	0	14	0	-	-	-	-	-	-	-	-	-	-
c1355	1574	8	117	0	-	-	-	-	-	-	-	-	-	-
c1908	1879	7	81	2	2	5	0	2	0.1	0	2	-	-	-
c2670	2747	98	132	19	8	243	11	19	99	0	8	0	4	7
c3540	3428	127	231	8	0	289	5	5	5	0	7	1	-	-
c5315	5350	59	453	0	-	-	-	-	-	-	-	-	-	-
c6288	7740	34	1231	10	0	295	3	0	23	0	7	3	-	-
c7552	7550	67	1045	64	0	3676	64	64	24	0	64	-	-	-
s5378	4090	39	189	0	-	-	-	-	-	-	-	-	-	-
s9234	6164	389	877	56	19	3449	17	54	73	0	22	28	0	6
s13207	8622	133	1320	21	15	1109	1	16	16	0	16	5	-	-
s15850	10263	374	2038	10	8	331	2	10	7	0	10	-	-	-
s35932	34144	3728	16112	0	-	-	-	-	-	-	-	-	-	-
s38417	27582	153	17401	8	4	1074	4	8	15	0	8	-	-	-
s38584	32125	1321	18932	24	22	3128	2	24	87	0	16	8	-	-

through both circuits using recursive learning to identify and store indirect implications. In phase 2, a test generator based on recursive learning is involved to justify  $E = 1$ . For more information, refer to [22].

In [23], a method using recursive learning is proposed to make any functional-based verification tool more efficient. Internal equivalencies found by recursive learning greatly aid in limiting the amount of resources used by functional approaches to logic verification. This method also provides a trade-off between structural and functional approaches to logic verification. Using recursive learning, functionally equivalent internal signal are extracted and Ordered Binary Decision Diagrams are formed, treating these internal equivalent signals as the pseudo-inputs. As demonstrated in [23], this greatly reduces the OBDD sizes required. In one of the ISCAS benchmark circuits, c3540, this hybrid approach produces better results in terms of CPU times then either of the

approaches. In some circuits such as c6288, which are known to consume exponential memory by functional methods, the structural phase of this hybrid method alone proves successful.

In [26], [24] a novel approach using recursive learning is proposed for multi-level logic optimization. Motivated by the observation in Section III-A, a general ATPG-based approach to logic optimization is currently developed deriving circuit transformations from implications. In [26] it is shown that implications can be used to determine for each circuit node those functions in the network with respect to which this node has only one cofactor. Furthermore, recursive learning, if performed for the 5-valued logic alphabet of Roth, can identify permissible functions for minimizing the circuit. In traditional synthesis techniques, an important issue is to perform "good" divisions. Our preliminary research has shown that recursive learning permits to identify good Boolean divisors that justify the effort to attempt a Boolean division.

TABLE X  
COMPARISON OF THE RECURSIVE LEARNING-BASED  
OPTIMIZATION TECHNIQUE WITH MIS

Circuit	Initial	MIS-II	Recursive Learning-based Method [26]
c1355	562	550	544
c1908	769	551	517
c2670	1023	741	718
c3540	1658	1253	1154
c432	270	196	161
c499	562	550	544
c5135	2425	1740	1697
c6288	3313	3313	3240
c7552	3087	2157	1855

Some preliminary results of this optimization on combinational benchmark circuits are listed in Table X, which gives a comparison between the number of literals present in the optimized circuit between MIS-II and the recursive learning-based method [26].

Furthermore, in [25] it was shown that recursive learning can also be used for optimizing sequential circuits. The method of [25] is based on adding and removing connections in the circuit. Recursive learning is used to identify permissible connections that can be added as well as for quick redundancy removal.

Recursive learning not only allows quick removal of redundancies, but it also gives valuable information on how to transform the circuit. Indirectness of implications closely relates to suboptimality in the circuit. Based on this observation, an approach to logic optimization proposed in [26] can be viewed as a generalization of the earlier proposed approach in [25].

## VII. CONCLUSION

We have presented a new technique called recursive learning as an alternative search method for test generation in digital circuits and with potential application to other CAD problems. Results clearly show that recursive learning is, by far, superior to the traditional branch-and-bound method based on a decision tree. This is a very promising result, especially if we keep in mind that recursive learning is a general new concept to solve the Boolean satisfiability problem; it therefore has potential for a uniform framework for development of both CAD and test algorithms. Recursive learning can, therefore, be seen under more general aspects. It is a powerful concept to perform a logic analysis on a digital circuit. By recursively calling the presented learning functions, it is possible to exact the entire situation of logic relations between signals in a circuit. This promises the successful application of recursive learning to a wide variety of problems, in addition to the test generation problem discussed in this paper.

Already, a recursive learning-based technique has provided fundamental insights in design verification problems [22], [23]. Current research also focuses on the application of recursive learning to logic optimization and other related procedures. First results for logic optimization [25], [26] are very promising. Further research of application of recursive learning to various CAD problems, therefore, is warranted.

## ACKNOWLEDGMENT

The authors are particularly grateful to Prof. Joachim Mucha, Head of *Institut für Theoretische Elektrotechnik*, Universität Hannover, Germany for his support of this work.

## REFERENCES

- [1] O. H. Ibarra and S. K. Sahni, "Polynomially complete fault detection problems," *IEEE Trans. Comput.*, vol. C-24, pp. 242-249, Mar. 1975.
- [2] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Trans. Comput.*, vol. C-30, pp. 215-222, Mar. 1981.
- [3] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," in *Proc. 13th Int. Symp. on Fault Tolerant Computing*, 1983, pp. 98-105.
- [4] T. Kirkland and M. R. Mercer, "A Topological Search Algorithm for ATPG," in *Proc. 24th Design Automation Conf.*, 1987, pp. 502-508.
- [5] U. Mahlstedt, T. Grüning, C. Özcan, and W. Daehn, "CONTEST: A Fast ATPG Tool of Very Large Combinational Circuits," in *Proc. Int. Conf. Computer Aided Design*, Nov. 1990, pp. 222-225.
- [6] M. Schulz, E. Trischler, and T. Sarfert, "SOCRATES: A highly efficient automatic test pattern generation system," in *Proc. Int. Test Conf.*, 1987, pp. 1016-1026.
- [7] M. Schulz and E. Auth, "Improved Deterministic Test Pattern Generation with Applications to Redundancy Identification," *IEEE Trans. Computer-Aided Design*, pp. 811-816, July 1989.
- [8] J. Rajski and H. Cox, "A Method to Calculate Necessary Assignments in Algorithmic Test Pattern Generation," in *Proc. Int. Test Conf.*, 1990, pp. 25-34.
- [9] P. Muth, "A Nine-Valued Logic Model for Test Generation," *IEEE Trans. Comput.*, vol. C-25, pp. 630-636, 1976.
- [10] S. B. Akers, "A Logic System for Fault Test Generation," *IEEE Trans. Comput.*, vol. C-25, no. 2, pp. 620-630, June 1976.
- [11] J. P. Roth, "Diagnosis of automata failures: A calculus & a method," *IBM J. Res. Develop.*, vol. 10, pp. 278-291, July 1966.
- [12] W. Kunz and D. K. Pradhan, "Accelerated Dynamic Learning for Test Generation in Combinational Circuits," *IEEE Trans. Computer-Aided Design*, vol. 12, no. 5, pp. 684-694, May 1993.
- [13] J. Giraldi and M. Bushnell, "Search State Equivalence for Redundancy Identification and Test Generation," in *Proc. Int. Test Conf.*, 1991, pp. 184-193.
- [14] T. Fujino and H. Fujiwara, "An Efficient Test Generation Algorithm Based on Search State Dominance," in *Proc. Int. Symp. Fault-Tolerant Comp.*, 1992, pp. 246-253.
- [15] T. Larrabee, "Efficient Generation of Test Patterns Using Boolean Difference," in *Proc. Int. Test Conf.*, 1989, pp. 795-801.
- [16] S. T. Chakradhar and V. D. Agrawal, "A Transitive Closure based Algorithm for Test Generation," in *Proc. 28th Design Automation Conf.*, 1991, pp. 353-358.
- [17] J. A. Waicukauski, P. A. Shupe, D. J. Giramma and A. Matin, "ATPG for Ultra-Large Structured Designs," in *Proc. Int. Test Conf.*, 1990, pp. 44-51.
- [18] W. Kunz and D. Pradhan, "Recursive Learning: An Attractive Alternative to the Decision Tree for Test Generation in Digital Circuits," in *Proc. Int. Test Conf.*, 1992, pp. 816-825.
- [19] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Designs and a Special Translator in Fortran," in *Proc. Int. Symp. on Circuits and Systems, Special Session on ATPG and Fault Simulation*, June 1985.
- [20] F. Brglez et al., "Combinational Profiles of Sequential Benchmark Circuits," *Int. Symp. on Circuits and Systems*, May 1989, pp. 1929-1934.
- [21] S. Kundu et al., "A Small Test Generator for Large Designs," in *Proc. Int. Test Conf. 1992*, pp. 33-40.
- [22] W. Kunz, "HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning," in *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 1993, Santa Clara, pp. 538-543.
- [23] S. Reddy, W. Kunz, and D. Pradhan, "Improving OBDD Based Verification using Internal Equivalencies," Dept. of Computer Science, Texas A&M Univ., College Station, Texas, Tech. Rep. 94-019, Jan. 1994, (submitted for publication).
- [24] W. Kunz and D. Pradhan, "Recursive Learning Technique and Applications to CAD," US Patent Application No. 08/263721.
- [25] L. Entrena and K. T. Cheng, "Sequential Logic Optimization by Redundancy Addition and Removal," *ICCAD 93*, Nov. 1993, pp. 310-315.
- [26] W. Kunz and P. Menon, "Multilevel Logic Optimization by Implication Analysis," *ICCAD '94*, to be published.



**Wolfgang Kunz** (S'90-M'91) was born in Saarbrücken, Germany, on February 7, 1964. From 1984 to 1989, he studied at the University of Karlsruhe, Germany, from which he received the Dipl.-Ing. degree in electrical engineering. During 1989, he was visiting scientist at the Norwegian Institute of Technology, Trondheim, Norway. In 1989, he joined the Department of Electrical and Computer Engineering at the University of Massachusetts, Amherst, where he worked as research assistant until August 1991. From October 1991 to March 1993,

he worked with Institut für Theoretische Elektrotechnik at the University of Hannover, Germany, where he obtained his Ph.D. in 1992. Since April 1993, he has been with Max-Planck-Society, Group for Fault-Tolerant Computing at the University of Potsdam, Germany. His research interests are in test generation, logic verification, logic optimization, and fault-tolerant computing. Dr. Kunz is a member of the IEEE and Verein Deutscher Elektrotechniker.



**Dhiraj K. Pradhan** (S'70-M'72-SM'80-F'88) holds the COE Endowed Chair in Computer Science at Texas A&M University, College Station, Texas. Prior to joining Texas A&M he served until 1992 as Professor and Coordinator of Computer Engineering at the University of Massachusetts, Amherst. Funded by NSF, DOD, and various corporations, he has been actively involved in VLSI testing, fault-tolerant computing and parallel processing research, presenting numerous papers, with extensive publication in journals over the last 20

years. Dr. Pradhan has served as guest editor of special issues on fault-tolerant computing of *IEEE Transactions on Computers* and *Computer*, published in April 1986 and March 1980, respectively. Currently, he is an editor for several journals, including *IEEE Transactions on Computers*, *Computer*, and *JETTA*. He has also served as the General Chair 22nd Fault-Tolerant Computing Symposium and as Program Chair for the IEEE VLSI test symposium. Also, Dr. Pradhan is a co-author and editor of *Fault-Tolerant Computing: Theory and Techniques*, Vols. I and II (Prentice Hall, 1986 and 1991). Dr. Pradhan is a Fellow of the IEEE and is a recipient of the Humboldt Distinguished Senior Award.