Recursive Learning: A New Implication Technique for Efficient Solutions to CAD-problems: Test, Verification and Optimization

Wolfgang Kunz*

Max-Planck-Society
Fault-Tolerant Computing Group
at the University of Potsdam
Am Neuen Palais 10
14415 Potsdam, Germany

Dhiraj K. Pradhan

Fault-Tolerant Computing Lab
Computer Science Department
Texas A&M University
College Station, TX 77843
email: pradhan@cs.tamu.edu

(appeared in *IEEE Transaction on Computer-Aided Design*, Vol. 13, No. 9, September 1994, pp. 1143 - 1158.)

* The preliminary research was initiated while both the authors were at the University of Massachusetts, Amherst and subsequently, essential parts of the reported research have been conducted while the first author was with Institut für Theoretische Elektrotechnik, University of Hannover, Germany. Research reported was supported, in part, by NSF MIP 92-18238.

Keywords: Recursive Learning, Unjustified Gates, Precise Implications, Necessary Assignments, Boolean Satisfiability, Design Verification

Abstract

Motivated by the problem of test pattern generation in digital circuits, this paper presents a novel technique called *recursive learning*, 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 a major progress in design verification problems. Specifically, one recursive learning-based technique was able to verify the formidable multiplier c6288 against the non-redundant version c6288nr in only 17seconds on a Sparc workstation 10/51 [22].

1. Introduction

The problem of test generation has been proven to be NP-complete [1]. Test generation for combinational circuits has been viewed [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], [4], [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 compter-aided circuit design such as logic verification and optimization [22 - 25]. Specifically, for the first time, it provides a uniform framework for *both* CAD and test problems. Using this, it has been shown [22] that the non-redundant initial MCNC versions were not equivalent to the original ISCAS 85 benchmarks -- a surprising result! 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. First results clearly show that recursive learning can be used to design very powerful techniques for multi-level logic optimization [24, 25].

In this paper, we motivate our new approach in the context of test generation. Along the process of generating a test vector for a given fault, some assignments of signal values are found to be *necessary*; i.e., they can be *implied* from the existing situation of value assignments in the circuit. 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 which 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, generates *all* necessary assignments.

Knowledge about necessary assignments is crucial for the number of backtracks which must be performed. Backtracks occur if wrong decisions have been made, decisions considered wrong if they violate necessary assignments. Hence, it is important to realize that if *all* necessary assignments are known at every stage of the test generation process, there can be no backtracks at all. All methods presented in the past, such as [5], [6], [7], [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 NP-complete and a method which guarantees identifying all necessary assignments must be exponential in time complexity.

This work develops a new method called *recursive learning* which can perform a complete search to identify all necessary assignments. The search for all necessary assignments, as opposed to the traditional search for one sufficient solution of the test generation problem, is of a totally different nature; recursive learning, therefore, represents a fundamentally new alternative alternative to *all* traditional techniques.

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 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 which 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]. The performance of recursive learning is evaluated here by applying it to the problem of test generation.

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] which allows for parallelization of test generation. It should be noted that these methods, unlike our technique, rely on a decision tree when conducting a complete search. Thus, they are likely to benefit from the searching scheme presented here.

2. 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 3 will be rooted in the observations made in the following example. Table 1 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 refered to as demo_recursive_learning() in Table 1. 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 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)

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 2, which lists the different learning implications performed 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 3. 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 learning 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. Now we proceed in the same way with unjustified line f_1, learning 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 include that: (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 above example is due to both unjustified lines 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, which 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 => 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 which can be accounted for as explained in Section 5. Nevertheless, procedure *demo_recursive_learning()* can miss necessary assignments because of incomplete forward implications. The reader may verify, as an exercise, that also, learning at line j will 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 to signal d, then the forward implication x=0 => 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 *unjustified gates*, on which recursive learning should be applied to make it identify *all* the necessary assignments.

3. 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 which result from previously made signal assignments. The concept of recursive learning allows to design methods able 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.

3.1 The Precise Implication Procedure

It was pointed out in the previous section that the concept of unjustified 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 Roth [11] , B_5=(0, 1, D, $\overline{D}$, X), a signal is specified, if it has one of the values '0', '1', 'D', or '$\overline{D}$'. 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 exists 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 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, 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, $J = \{f_1=V_1, f_2=V_2, ... f_n=V_n\}$, where $f_1, f_2, ... f_n$ 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 $J_1, J_2, ... J_m$ for an unjustified gate G. If there is at least one justification $J_i \in {}^{G}C$, i=1,2...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 signal 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 which have to be made for all justifications that represent a complete set of justifications, also have to be made for any other 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 which 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 which 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 ....} is the set of all unjustified gates as they result from the current state of the test generation algorithm.

${}^{Gx}J^{r}$ = { f_1=V_1, f_2=V_2,.... } is a set of assignments that represents a justification for some gate G_x in a given recursion level r.

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

${}^{Jx}U^{r}$ = { G_1, G_2, G_3,...} 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 3 depicts procedure *make_all_implications()*, 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 3 makes precise implications; i.e., a finite r_{max} always exists such that *make_all_implications(0,*r_{max}*)* determines all necessary assignments for a given set of unjustified gates, ${}^{0}U$.

Proof:

Preliminary Remarks: Making precise implications means to identify all signal values, which are uniquely determined due to the unjustified gates contained in the set ${}^0U$. 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 ${}^{Gx}J_i^r \in {}^{Gx}C^r$:

Case1: The direct implications for the set of assignments J_i at G_x yield f=V

Case2: The assignment f=V is necessary for the justification of at least one gate in the set of unjustified gates ${}^{Ji}U^{r+1}$.

In the first case, the necessary assignment is recognized and learned when learning in level r+1. In the latter, deeper recursion levels 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 ${}^0U$, the recursive call of *make_all_implications()* will always reach a level *max,* such that ${}^{Ji}U^{max} = \emptyset$ for all justifications $J_i \in {}^{Gx}C^{max-1}$ 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 ${}^{Gx}C^{n-1}$, procedure make_all_implications() 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 justifications ${}^{Gx}C^{n-1}$.) The above is true for any gate $G_x \in {}^{Ji}U^{n-1}$, where J_i is some justification J_i^{n-2} for some gate in U^{n-2}. Hence, all necessary 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 ${}^{0}U$.

q.e.d

Fig. 5 shows some combinational circuitry to illustrate *make_all_implications*. Table 4 lists the single steps to perform the implication p=1 => q=1. Note that the learning techniques [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 sets 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. In order 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 e.g. 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 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 by 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 Fig. 1, Fig. 2 and Fig. 5: Necessary assignments are only missed by direct implications because the shown circuits contain sub-optimal circuitry. In the scheme of Fig. 6 the intersections 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 observation has motivated the research reported in [24], [25].

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 *precise_implications(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. 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. Think carefully why this must be true. 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 step 2 and 3 of the complete induction are straightforward.

3.2 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 which are due to the requirement of propagating the fault signal to at least one primary output. In analogy 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 which have a faulty value and a successor signal which 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 signal a we obtain F^0={b, e, g, h}.

Table 5 lists procedure *fault_propagation_learning()*. In Table 6, it is explained step by step, how *fault_propagation_learning* is used to determine the necessary assignment n=0 in Fig. 7.

Procedure *fault_propagation_learning()* which calls procedure *make_all_implications()*, correctly learns all assignments which 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 which is sensitized in procedure *fault_propagation_learning()*, gates become unjustified, if there is more than one possibility to sensitize them. This is demonstrated in Table 6 for gate G in Fig. 7.

Procedure *fault_propagation_learning()* as given in Table 5 does not show how to handle XOR- gates. However, the extension to XOR- gates is straightforward. XOR- gates as well as 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.

4. Test Generation with Recursive Learning

4.1 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 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 as we wish. Note that this allows attractive possibilities for new heuristics. In order 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 in order not to spend a lot of time on learning operations. Only if the precision is not sufficient to avoid wrong decisions, it is sensible to increment r_{max}.

There are many possibilities to choose r_{max}. In order 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 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 non-solution 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 if the precision has to be increased any further or not. It also makes sure that we can never enter the

same non-solution 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 non-solution areas of the search space as with the original FAN- algorithm. The point of comparison is how fast the non-solution 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 may initially traverse very deep into non-solution 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 incremented recursion depth first only to the situation of value 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.

4.2 Compatibility and Generality of the Approach

A lot of 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. In the same way as they reduce the number of backtracks in the decision tree, they 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 due to the fact that 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 "working horse" of recursive learning. Finally, it is possible to use recursive learning and the decision tree at the same time to combine the advantages 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 which is unjustified may have a lot of justifications. However note, that we only have to keep trying 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, a lot of justifications have to be tried only, if there are actually common values for many justifications.

It is beyond the scope of this paper to examine the application of recursive learning in other fields such as design verification and logic synthesis [22 - 25]. However, since

recursive learning is general scheme to solve Boolean satisfiability, it promises to be useful in many different applications. Specifically, excellent results using recursive learning have been already reported [22-25].

4.3 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 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. In analogy to 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 2, 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 does neither have to derive all sufficient solutions explicitly nor implicitly in order 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 which are necessary in the current level of recursion are retained. With the decision tree, however, all assignments which are not proved 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 a lot of enumeration is necessary with the decision tree until the wrong decision is finally reversed. For the precise implication however, a lot of searching is only 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 current research.

5. Experimental Results

In order to examine the performance of recursive learning we use the FAN- algorithm. Our goal is not to present a new tool for test generation but to compare the performance of two searching schemes. For comparison, we use the original FAN- algorithm with the decision tree and 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.

There are two general aspects of recursive learning in a FAN- based environment which we use for better efficiency: first, as discussed in section 2, 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 bigger than 5. Otherwise, no learning is performed for such gates. '5' was chosen intuitively to suppress the unnecessary recursions so that they contribute only 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 we also perform *fault_propagation_learning()*.

In order to illustrate the different nature of the two searching schemes, we first compare recursive learning to the traditional search, by only looking at redundant faults. In our first experiment, we only target all redundant faults in the ISCAS85 [19] benchmarks and the seven largest ISCAS89 [20] benchmarks. The results are given in Table 7. The first column lists the circuit which 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. Column 6 and 7 give the CPU-time and the number of aborted faults for recursive learning. The next 4 columns show for how many faults which 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.

The results impressively show the superiority of recursive learning for redundancy identification when compared to the decision tree. Look for example at circuit c3540: With the decision tree 5 faults are aborted after performing 1000 backtracks each. There is 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" which is typical for redundancy identification with the decision tree. If the implications fail to reveal the conflict the search space has to be exhausted, in order 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 are the cases 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. In those cases it is faster to perform a few backtracks with the decision tree as we have shown already 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 splitted into two groups of 10 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 time we explore the search space in orthogonal direction compared to our first attempt. For testable faults this procedure has shown to be very effective. As an example, in circuit c6288, if test generation is performed for all faults, 225 faults have remained undetected after the first ten backtracks. After the following ten backtracks with orthogonal heuristics, only ten faults were left. Faults which 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 8 shows the results of test generation for the ISCAS85 benchmarks and for the 7 largest ISCAS89 benchmarks. After each generated test vector, fault simulation is used to reduce the faultlist (faultdropping). No random vectors are used. The first two columns list the circuits under consideration and the number of faults which 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 which are identified 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 7.

Table 9 shows the results if all faults are targeted. There is neither a random phase nor faultdropping.

The results in Table 8 and Table 9 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 few exceptions). The results show that recursive learning can replace the "whole bag of tricks" which 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 on the examination of a new searching method and not on presenting 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 which 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 Mbyte, which is usually negligible.

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 extract 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 as provided fundamental progress 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 [24, 25] are very promising.

Acknowledgments

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] Ibarra O.H., Sahni S.K.: "Polynomially complete fault detection problems", IEEE Transactions on Computers, vol. C24, March 1975, pp. 242-249.

[2] Goel P.: "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits", IEEE Transactions on Computers, vol. C-30, March 1981, pp. 215-222.

[3] Fujiwara H., Shimono T.: "On the Acceleration of Test Generation Algorithms", Proc. 13th Int. Symp. on Fault Tolerant Computing, 1983, pp. 98-105.

[4] Kirkland T., Mercer M. R.: "A Topological Search Algorithm for ATPG", Proc. 24th Design Automation Conf., 1987, pp. 502-508.

[5] Mahlstedt U., Grüning T., Özcan C., Daehn W.: "CONTEST: A Fast ATPG Tool for Very Large Combinational Circuits", Proc. Intl. Conference on Computer Aided Design, Nov. 1990, pp. 222-225.

[6] Schulz M., Trischler E., Sarfert T.: "SOCRATES: A highly efficient automatic test pattern generation system", Proc. Intl. Test Conf., 1987, pp.1016-1026.

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

[8] Rajski J., Cox H.: "A Method to Calculate Necessary Assignments in Algorithmic Test Pattern Generation", Proc. Intl. Test Conf., 1990, pp.25-34.

[9] Muth P.: "A Nine-Valued Logic Model for Test Generation", IEEE Trans. on Computers, vol. C-25, 1976, pp. 630-636.

[10] Akers S.B.: "A Logic System for Fault Test Generation", IEEE Trans. on Computers, vol. C-25, no. 2, June, 1976, pp. 620- 630.

[11] Roth J. P.: "Diagnosis of automata failures: A calculus & a method", IBM J. Res. Develop., vol. 10, July 1966, pp. 278-291.

[12] Kunz W., Pradhan D.K.: " Accelerated Dynamic Learning for Test Generation in Combinational Circuits", IEEE Trans. on Computer-Aided Design, vol. 12, no. 5, May 1993, pp. 684-694.

[13] Giraldi, J., Bushnell M.: "Search State Equivalence for Redundancy Identification and Test Generation", Proc. Intl. Test Conf., 1991, pp. 184-193.

[14] Fujino T., Fujiwara H.: "An Efficient Test Generation Algorithm Based on Search State Dominance", Proc. Intl. Symp. on Fault-Tolerant Comp., 1992, pp. 246-253.

[15] Larrabee T.: "Efficient Generation of Test Patterns Using Boolean Difference", Proc. Intl. Test Conf., 1989, pp. 795-801.

[16] Chakradhar S.T., Agrawal V.D.: "A Transitive Closure based Algorithm for Test Generation", Proc. 28th Design Automation Conf., 1991, pp. 353-358.

[17] Waicukauski J.A., Shupe P.A., Giramma D.J., Matin A.: "ATPG for Ultra-Large Structured Designs", Proc. Intl. Test Conf., 1990, pp. 44-51.

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

[19] Brglez F., Fujiwara H.: "A Neutral Netlist of 10 Combinational Benchmark Designs and a Special Translator in Fortran", Proc. Intl. Symp. on Circuits and Systems, Special Session on ATPG and Fault Simulation, June 1985.

[20] Brglez F. et al.: "Combinational Profiles of Sequential Benchmark Circuits", Intl. Symp. on Circuits and Systems, May 1989, pp. 1929-1934.

[21] Kundu S. et al.: "A Small Test Generator for Large Designs" Proc. Intl. Test Conf. 1992, pp. 30-40.

[22] Kunz W.: "HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning", Proc. Intl. Conf. on Computer-Aided Design (ICCAD), Santa Clara, Nov. 1993, pp. 538-543

[23] Reddy S., Kunz W. and Pradhan D.: “Improving OBDD Based Verification using Internal Equivalencies”, Technical report 94-019, Department of Computer Science, Texas A&M University, College Station, Texas Jan. 1994 (submited for publication).

[24] Kunz W.: “Ein neuer Ansatz für die Optimierung mehrstufiger logischer Schaltungen”, Proc. GI/GME/ITG-Fachtagung Rechnergestützter Entwurf und Architektur mikroelektronischer Systeme, Oberwiesenthal, Germany, May 1994.

[25] Kunz W., Menon P.: “Multi-level Logic Optimization by Implication Analysis”, submitted for publication.

List of Tables

Table 1: Preliminary learning routine

Table 2: Using *demo_recursive_learning()*

Table 3: Precise implication procedure

Table 4: Demonstrating procedure *make_all_implications()*

Table 5: Procedure *fault_propgagation_learning()*

Table 6: Demonstrating *fault_propagation_learning()*

Table 7: Experimental results for redundant faults (Sun SPARC 10/51)

Table 8: Experimental results for test generation with faultdropping (Sun SPARC 10/51)

Table 9: Experimental results for test generation without faultdropping (Sun SPARC 10/51)

```
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
  }
}
```

Table 1: Preliminary learning routine

0. learning level	1. learning level	2. learning level
(generally valid		
signal values)		unjust. line e1 = 0:
i1 = 0 (unjust.)	unjust. line i1 = 0:	1. justif.: a1 = 0
j = 1 (unjust.)		=> a2= 0
	1. justif.: g1 = 0	=> **e2 = 0**
enter	=> e1= 0 (unjust.)	
learning ->	=> f1= 0 (unjust.)	2. justif.: b1 = 0
	enter next	=> b2 = 0
	recursion ->	=> **e2 = 0**
	e2 = 0	<======
		unjust. line f1 = 0:
		1. justif.: c1 = 0
		=> c2 = 0
		=> **f2 = 0**
		2. justif.: d1 = 0
		=> d2 = 0
		=> **f2 = 0**
	f2 = 0	<======
	=> g2 = 0	
	=> **i2 = 0**	
	=> **k = 1**	
	2. justif.: h1 = 0	
	=> h2 = 0	
	=> **i2 = 0**	
	=> k = 1	
k = 1	<======	
i2 = 0		
	unjust. line j = 1:	
	...	

Table 2: Using *demo_recursive_learning*

initially: $r=0$;

make_all_implications(r, r_{max})

```
{
  make all direct implications and set up a list U^r of
  resulting unjustified gates
  if r<r_max : learning
   {
    for each gate G_x, x=1,2.., in U^r: justifications
     {
        set up list of justifications GxC^r
        for each justification J_i ∈ GxC^r:
         {
          -  make the assignments contained in J_i
          -  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 J_i ∈ GxC^r 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
     }
   }
}
```

Table 3: Precise implication procedure

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

Table 4: Demonstrating procedure *make_all_implications()*

```
fault_propagation_learning(r, r_max)
{
  for all signals f_D ∈ 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, r_max);
    set up list of new D- frontier F^(r+1);
    if ( r< r_max  and current sensitization is consistent )
    fault_propagation_learning(r+1, r_max);
   }
  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)
}
```

Table 5: Procedure *fault_propagation_learning()*

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

Table 6: Demonstrating *fault_propagation_learning()*

Results for collapsed faultlist		1. PHASE (eliminate easy faults)			2. PHASE for DIFFICULT FAULTS (aborted in 1. phase)									
no faultdropping, *all* faults are targeted		FAN with backtrack limit of 10+10			FAN with DECISION TREE (bt. limit of 1000)			FAN with RECURSIVE LEARNING			learning levels:			
circuit	no. faults targeted	red.	time [s]	aborted	red.	**time [s]**	**ab.**	red.	**time [s]**	**ab.**	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	**98**	**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**	**37**	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	-	-

Table 9: Experimental Results for test generation without faultdropping (Sun Sparc Workstation 10)

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			learning levels:			
circuit	no. faults targeted	red.	time [s]	aborted	red.	**time [s]**	**ab.**	red.	**time [s]**	**ab.**	r1	r2	r3	r4
c432	93	1	1	3	0	**12**	**3**	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	**98**	**0**	8	0	4	7
c3540	392	127	47	5	0	**278**	**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	**3449**	**37**	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	**3128**	**2**	24	**87**	**0**	16	8	-	-

Table 8: Experimental Results for test generation with faultdropping (Sun Sparc Workstation 10)

Results if only redundant faults are targeted		FAN with DECISION TREE (bt. limit of 1000)			FAN with RECURSIVE LEARNING learning levels:						
circuit	no. faults targeted	no. of backtracks	**time [s]**	**ab.**	**time [s]**	**ab.**	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	**3858**	**64**	**36**	**0**	65	66	-	-	-
s5378	39	0	**1**	**0**	**1**	**0**	39	-	-	-	-
s9234	443	55396	**4235**	**35**	**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	-	-

Table 7: Experimental results for redundant faults (Sun Sparc Workstation 10)

List of Figures

Fig. 1: Circuitry to demonstrate recursive learning

Fig. 2: Incomplete forward implications

Fig. 3: Justifications for unjustified gates

Fig. 4: Determining a complete set of justifications

Fig. 5: Making precise implications for p=1

Fig. 6: Schematic illustration of recursive learning

Fig. 7: Necessary assignments for fault propagation

Fig. 8: Algorithm for choosing r_{max}

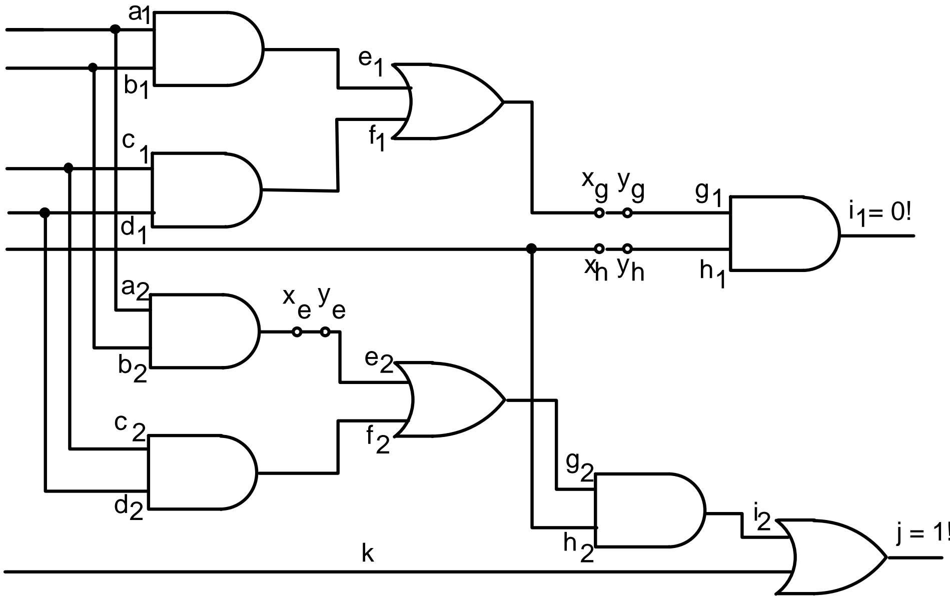

Fig. 1: Circuitry to demonstrate recursive learning

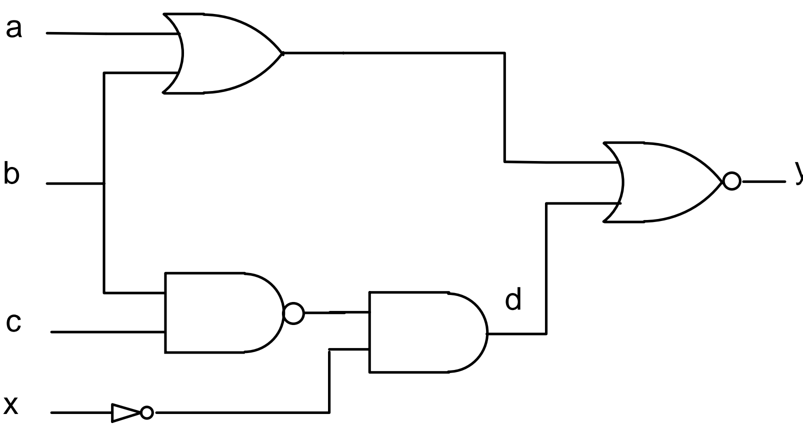

Fig. 2: Incomplete forward implications

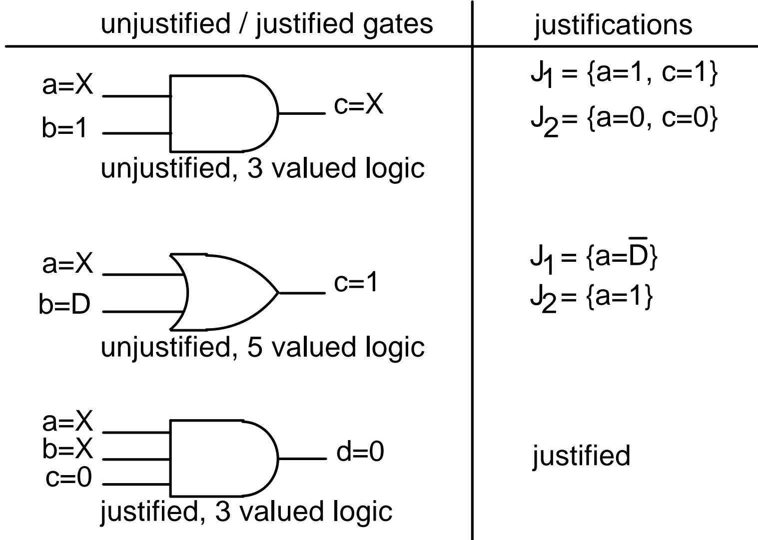

Fig 3: Justifications for unjustified gates

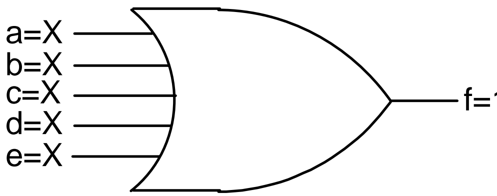

Fig. 4: Determining a complete set of justifications

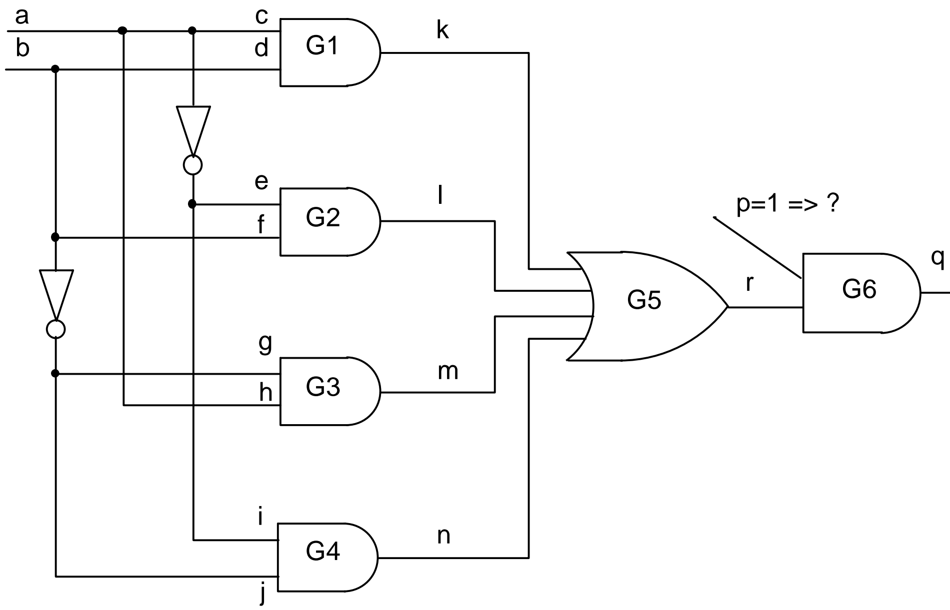

Fig. 5: Making precise implications for p=1

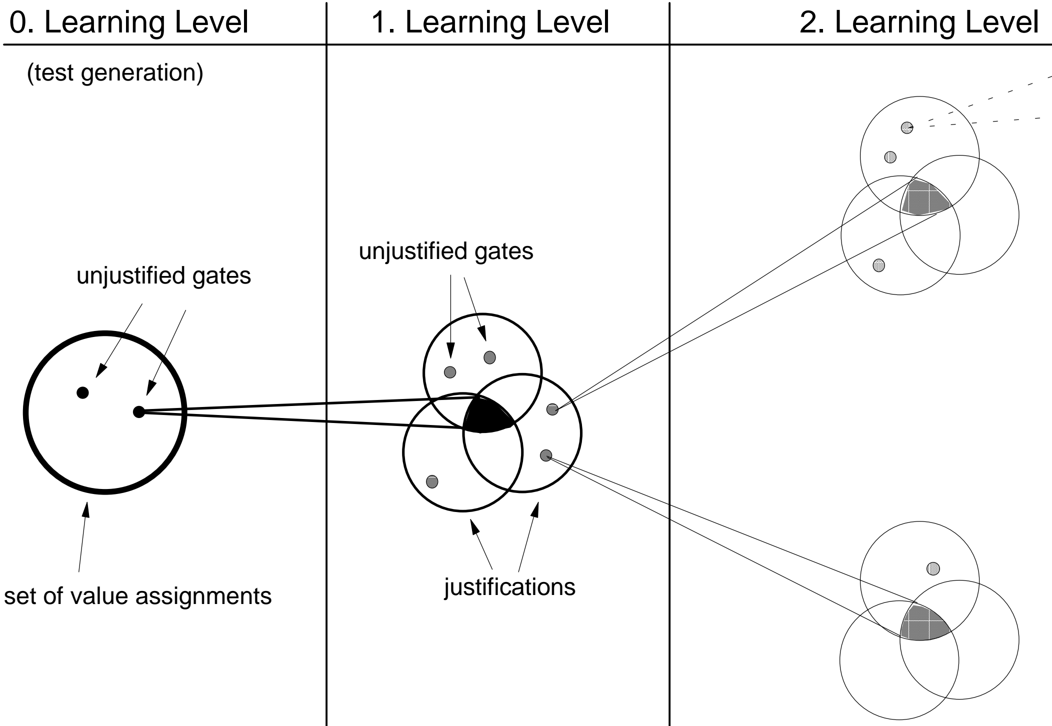

Fig. 6: Schematic illustration of recursive learning

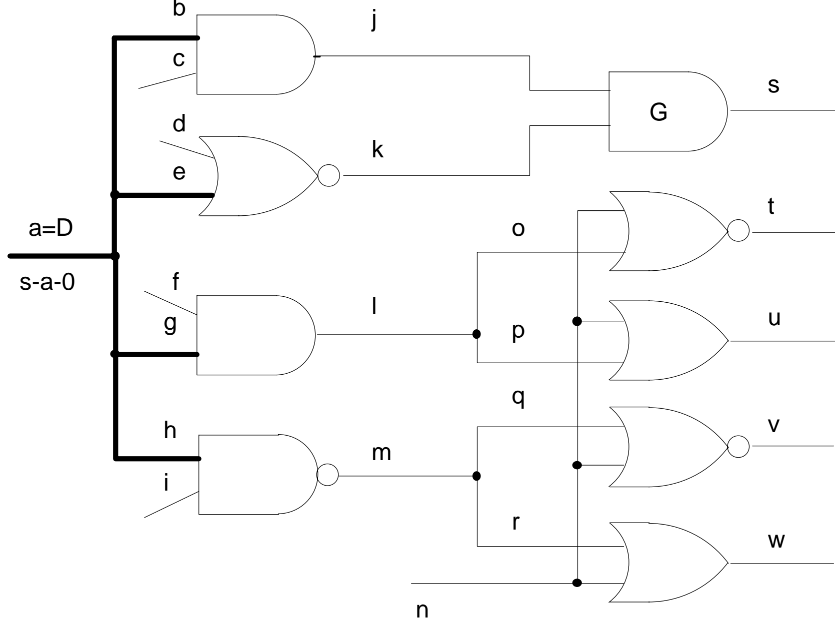

Fig. 7: Necessary assignments for fault propagation

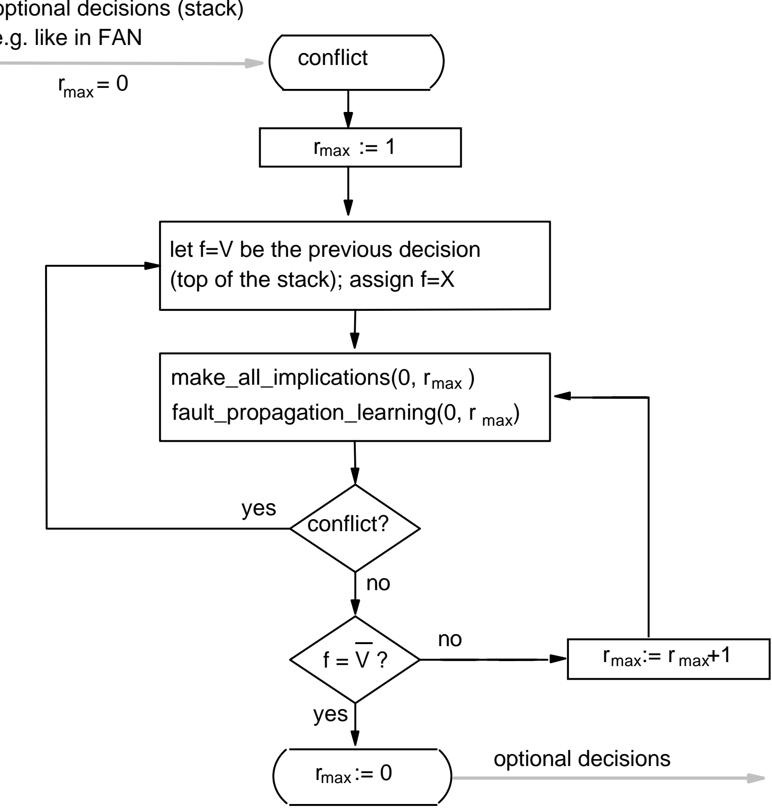

Fig. 8: Algorithm for choosing r_{max}

Biography

Wolfgang Kunz was born in Saarbrücken, Germany, in 1964. From 1984 to 1989, he studied at the University of Karlsruhe, Germany where 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 is 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 member of IEEE and Verein Deutscher Elektrotechniker.

Dhiraj K. Pradhan is holder of 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 publications in journals over the last twenty 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 and Computer* and *JETTA*. He has also served as the General Chair of the 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 the book entitled, *Fault-Tolerant Computing : Theory and Techniques,* Vols. I and II (Prentice Hall, 1986; @nd ed., 1991).

Dr. Pradhan is a Fellow of IEEE and is a recipient of the Humboldt Distinquished Senior Award.