

Recursive Learning

Madhurima Maddela, *Student Member, IEEE*

Auburn University, Auburn, AL 36849, USA

Abstract —Previous test generators for combinational and sequential circuits used a decision tree to systematically explore the search space while trying to generate a test vector. Recursive learning was introduced as an interesting alternative. Using recursive learning, 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. But its applications were not just restricted to test vector generation. This method helped make significant progress in redundancy identification, logic verification, boolean satisfiability and logic synthesis to name a few areas. This paper attempts to describe the concept in detail and illustrate it with a few examples.

Index Terms — Recursive Learning, Boolean satisfiability.

I. INTRODUCTION

Recursive learning can be understood as a general method to conduct a logic analysis deriving a maximum amount of information about a logical circuit in a minimum amount of time. First introduced in [1] and [2] and further developed in [3] and [4], *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 known.

The problems of test generation, verification and optimization are inter-related [5] - [6] and *NP complete* [7]. Test generation for combinational circuits has been viewed as an 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. 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. Recursive learning presents a new technique that can handle these pathological cases in test generation much more efficiently than the traditional search.

Boolean satisfiability (SAT) is intrinsic to many problems in *Electronic Design Automation* (EDA). In most cases, the problem formulation starts from a circuit description, for a given (circuit) property that needs to be validated for at least one primary input vector. The resulting circuit formulation, which may only be implicitly specified, is then mapped into an instance of SAT, in most cases using *Conjunctive Normal*

Form (CNF) formulas. This allows the usage of existing and extensively validated algorithms instead of dedicated ones. But mapping can take up a good percentage of the running time and computed input patterns are in general over specified. These problems can be overcome using recursive learning.

Section II describes the concept of recursive learning in good detail. Section III gives an insight into combinational test pattern generation with an example. Section IV describes an example to illustrate the use of recursive learning in CNF formulae. Section V briefly covers other different promising applications of recursive learning to verification and synthesis problems.

II. OVERVIEW OF RECURSIVE LEARNING

While performing logic synthesis of a circuit, some assignments of signal values are found to be *necessary*; i.e. those assignments can be implied in any way 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; however, all these techniques are limited in that they fail to produce *all* necessary assignments. Recursive learning can provide for this completeness. For example, traditional test generation is a search for one sufficient solution whereas recursive learning may be viewed as searching for those conditions that enable purging the non-solution area from the search space.

Recursive learning is not restricted to any particular logic alphabet and can be used for any logic value system such as five-valued logic, nine-valued logic or sixteen-valued logic. Also, the 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 this method is exponential in the maximum depth of recursion r_{max} . Memory requirements however grow linearly with r_{max} . As noted earlier, any methods that identify all necessary assignments must be exponential in time complexity because this problem is NP complete.

Some formal definitions of terms are presented below. Def. 1 uses a common notation of a *specified signal* by which we understand a signal with a fixed value. In a five-valued logic, $B_5 = (0, 1, D, \bar{D}, X)$, 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. [4]

Def.1: Given a gate G that has at least one specified input or output signal. Gate G is called *unjustified* if there is at least one or several unspecified input or output signal combinations 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 $f = V$, 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.

Unjustified gates describe the locations *where* learning has to be performed. The next definition determines *what* signal values have to be injected at the unjustified gates in order to perform learning.

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.

At every unjustified gate it is necessary to examine the logical consequences for a complete set of 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 signal assignments that represent a justification of the considered gate. Often the set can be smaller. All learning operations rely on a basic implication technique. These basic implications are called *direct implications*. A well-known example of direct implications in combinational circuits is the implications as performed in FAN [9]. Direct implications are performed in an event-driven way by evaluating the truth tables of gates that have an event and by propagating the signal values according to the connectivity in the circuit. For example, if the output of an AND gate is 1, then from the truth table we see that the only possibility of that event is when all the inputs to the AND gate are 1. *Indirect implications* are performed by learning. Learning means justifications for unjustified gates are temporarily injected in order to examine their logical consequences. By performing this process recursively, it is possible to determine all necessary assignments.

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} . Intuitively, a lot of recursions are only needed if there is a lot of redundant

circuitry. As we will see later in examples, at some point the justifications must reach the primary inputs and output so that no new unjustified gates requiring further recursions can be caused. In most cases, a recursion depth of 5 is about as high as r_{max} will go to.

There are many possibilities for choosing r_{max} . Figure 1 shows one such algorithm [11]. The idea behind the routine is that we use learning only to leave the non-solution areas as quickly as possible. If a conflict arises, the previous decision will be erased. If there is no conflict, it 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 to be 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 non-solution area twice. And the algorithm guarantees the completeness of test generation and redundancy identification without the use of a decision tree.

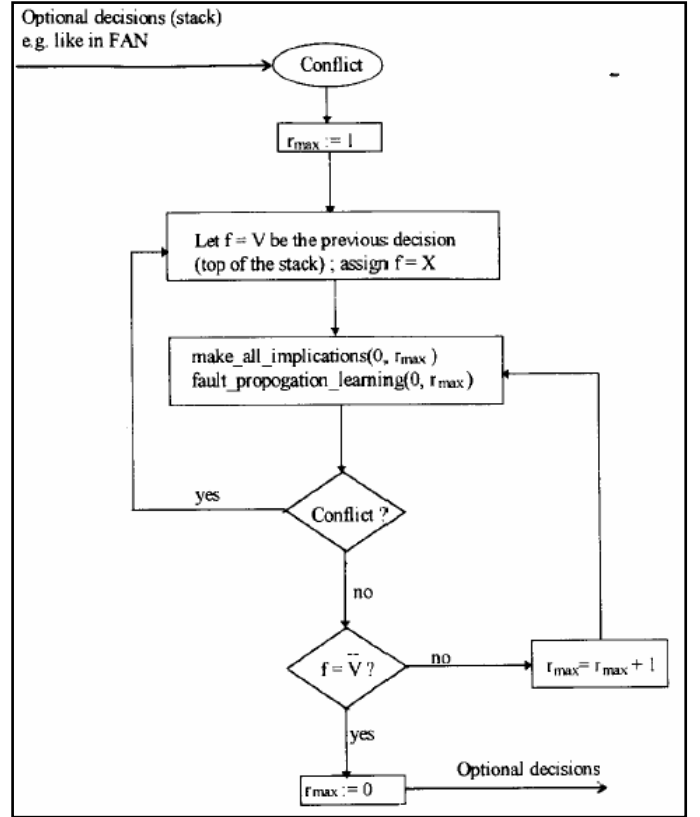


Fig. 1. Algorithm for choosing r_{max} [11]

III. DEMONSTRATING FAULT PROPAGATION LEARNING

In principle, the problem of test generation is solved using precise implication techniques. Observability constraints can always be expressed in terms of unjustified lines by means of Boolean difference. However, most ATPG algorithms use the

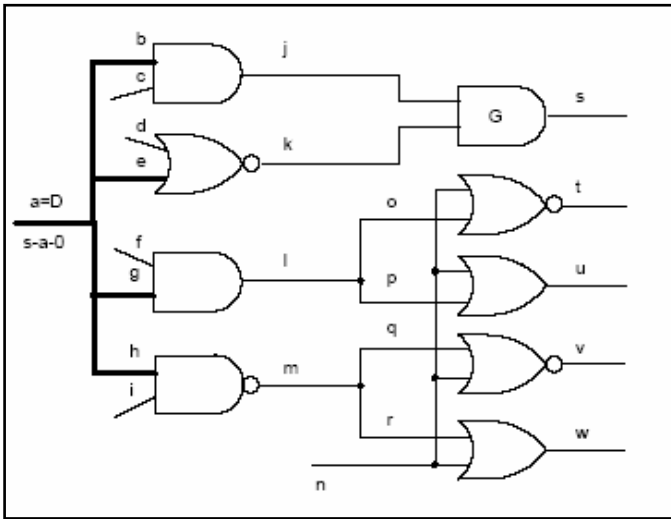


Fig. 2. An example for D-frontier. [11]

```

fault_propagation_learning(r, r_max)
{
  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, 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)
}

```

Fig. 3. An algorithm for fault propagation learning. [11]

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

Fig. 4. Fault propagation using recursive learning[11]

concept of a D-frontier. This makes it easier to consider topological properties of the circuit. This section shows how recursive learning can derive all conditions for fault propagation by injecting *sensitizations* [8] 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. Figure 2 show 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 $F^0 = \{b, e, g, h\}$. Figure 3 lists the procedure employed. 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 the *fault_propagation_learning()* procedure, gates become unjustified if there is more than one

possibility to sensitize them. It will be seen shortly that gate G , falls in this category.

Figure 4 explains the recursive learning process in detail. The aim of this example is to propagate the stuck-at-0 fault of a through b, e, g, h to any one of the outputs s, t, u, v, w . This is shown in Column 1 of ‘0 learning level’. Each of the D-frontier signals b, e, g, h are tested in the ‘learning level 1’, Column 2. Let us first examine if the path from b to s will ensure the propagation of the fault. Since $b = D$, the only way this fault will be propagated through an AND gate is when c is 1. This makes $j = D$. If this fault is successfully propagated, then $s = D$. This possibility is studied in Column 3, ‘learning level 2’. Here we see that for $s = D$ when $j = D$, k must be 1, but k cannot be 1 because $e = D$. So, due to this inconsistency, the sensitization fails for gate G . This information is sent back to Column 2.

Now, the D-frontier signal e is sensitized in Column 2. This sensitization also fails since an inconsistency develops when j needs to be set to 1. This is not possible since $b = D$. The failure of this sensitization is passed on to Column 2 which then proceeds with the next D-frontier signal g .

For the fault $g = D$ to be propagated to l, f must be 1. Once that is taken care of, for the next successive sensitizations, this enters ‘learning level 2’. It turns out that $l = D$ propagates to t when $n = 0$. Also, for $l = D$ to show up at u , the sensitization required is $n = 0$. Therefore, the only necessary assignment for the fault to be propagated has been learnt as $n = 0$.

This example underscores two important phenomena - the learning procedure can bring itself out of the non-solution area of the search space and it can also successfully pick out the indirect implications necessary for the process.

IV. SOLVING SATISFIABILITY IN COMBINATIONAL CIRCUITS

SAT is a widely used modeling tool in EDA. It finds application in test pattern generation for Stuck-at faults, delay-fault testing, redundancy removal, logic synthesis and equivalence checking and circuit delay computation among many other problems. Given a Boolean function ψ with n variables, the SAT problem consists of assigning values to the variables such that ψ assumes the value 1, or proving that no such assignment exists and the function is equal to 0. A Boolean product of sums, or CNF formula [12], is the most often used representation of the Boolean function. A CNF formula is a conjunction of clauses, each of which is a disjunction of literals. A literal is either a variable or its negation. For example, CNF formula $\psi = (a + \bar{b})(b + \bar{c} + d)$ contains two clauses $(a + \bar{b}), (b + \bar{c} + d)$ and five literals. A clause is *satisfied* if at least one of its literals assumes value 1, *unsatisfied* if all its literals assume value 0, *unit* if all but one literal assume value 0, and *unresolved* otherwise. Literals with no assigned boolean value are *free literals* [13]. A formula is satisfied if all its clauses are satisfied and unsatisfied if at least

one clause is unsatisfied. It is often simpler to refer to clauses as sets of literals, and to the CNF formula as a set of clauses.

Figure 5 gives an example of a depth-2 recursive-learning procedure on CNF formulas. We can initiate the recursive-learning process on ω_7 because $y = 1$. Assignments $c = 0$ and $f = 0$ satisfy clause ω_7 . The recursive-learning process for depth 1 considers each of these assignments separately. Assignment $c = 0$ implies no assignments. However, clause ω_3 now exhibits a new literal with value 0. Hence, at depth 2 of the recursive learning process, we analyze clause ω_3 . Assignments $a = 0$ or $b = 0$ can satisfy ω_3 . Considering each assignment individually yields the implied assignment $x = 0$. For $a = 0$, we get $x = 0$ from clause ω_1 ; for $b = 0$, we get the implied assignment from ω_2 because $v = 0$.

We perform a similar analysis for $f = 0$. Because $f = 0$, clause ω_6 exhibits a literal with value 0. Hence, we analyze ω_6 at depth 2 and find that assignments $d = 0$ and $e = 0$ can satisfy it. Both $d = 0$ and $e = 0$ imply assignment $x = 0$. For $d = 0$, we get $x = 0$ from clause ω_4 ; for $e = 0$, we get $x = 0$ from ω_5 because $u = 1$.

Hence, $(y = 1) \wedge (v = 0) \wedge (u = 1) \rightarrow (x = 0)$; or in clausal form, $y + \bar{v} + u + \bar{x}$. Thus, the recursive-learning process identifies a new clause that implies assignment $x = 0$.

Assignments: $\{y = 1, u = 1, v = 0\}$

$$\omega_1 = (\bar{x} + a)$$

$$\omega_2 = (v + \bar{x} + b)$$

$$\omega_3 = (\bar{a} + \bar{b} + c)$$

$$\omega_4 = (\bar{x} + d)$$

$$\omega_5 = (\bar{u} + \bar{x} + e)$$

$$\omega_6 = (\bar{d} + \bar{e} + f)$$

$$\omega_7 = (\bar{c} + \bar{f} + y)$$

$$\omega_8 = (x + w + \bar{z})$$

Fig. 5. Recursive learning on a CNF formula. [14]

When solving SAT on a CNF formula derived from a combinational circuit for which we have structural information (using the additional layer described earlier), we can improve the recursive-learning procedure’s performance by restricting it to unjustified node assignments. In addition, for each unjustified node assignment, $y = v_y$, we can restrict the reasoning procedure to unresolved clauses that relate y to immediate fan-in nodes. This approach yields a significantly faster recursive-learning procedure, similar to the original procedure for circuits, and can also identify implicates of the CNF formula.

V. OBSERVATIONS

Figure 6 helps get a grasp of the actual values of computation times, recursion depths, etc. It shows a comparison between FAN algorithm using decision tree and recursive learning methods. The decision tree method has an upper limit on backtracks of 1000 per fault while the recursive learning method has an r_{\max} of 5. We can observe that for small circuits like c1355 and c5315, both perform similarly. But in other cases, recursive learning does not have any aborted faults, uses much less computation time and rarely reaches r_{\max} .

It is not wise to use recursive learning where many solutions exist that are easy to find. For such cases, it is faster to perform a few backtracks with the decision tree.

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	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	-	-

Fig. 6. Comparison of decision tree and recursive learning [11]

VI. SUMMARY

Recursive learning is a powerful technique toward identifying indirect implications between two circuits under consideration, when they exist. It has a wide range of applications including test pattern generation, boolean satisfiability, combinational equivalence checking and logic synthesis. The concept has been described with examples from a couple of applications. It is important to note that recursive learning is not limited to combinational circuits alone. [15] describes a method similar to FIRES for sequential circuits to identify redundant and untestable vectors.

REFERENCES

- [1] M. Schulz, E. Trischler, and T. Sarfert, "SOCRATES: A highly efficient automatic test pattern generation system," *Proc. Int. Test Conf.*, pp. 101&1026, 1987.
- [2] M. Schulz and E. Auth, "Improved Deterministic Test Pattern Generation with Applications to Redundancy Identification," *IEEE Trans. Computer-Aided Design*, pp. 811-816, 1989.
- [3] 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, 1993.
- [4] W. Kunz, D. K. Pradhan, "Recursive Learning: A Precise Implication Procedure and its Application to Test Generation in Digital Circuits", *IEEE Trans. on Computer-Aided Design of integrated circuits and systems*, vol. 13, no. 9, 1994.
- [5] W. Kunz, "HANNIBAL: An Efficient Tool for Logic verification Based on Recursive Learning," *Proc. Int. Conj: Computer-Aided Design (ICCAD)*, pp. 538-543, 1993.
- [6] W. Kunz and D. K. Pradhan, "Recursive Learning Technique and Applications to CAD," *US Patent Application No. 08/263721*.
- [7] H. Ibarra and S. K. Sahni, "Polynomially complete fault detection problems," *IEEE Trans. Comput.*, vol. C-24, pp. 242-249, 1975.
- [8] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE TranJ. Comput.*, vol. C-30, pp. 215-222, 1981.
- [9] H. Fujiwara, T. Shiono, "On the Acceleration of Test Generation Algorithms", *13th Intl. Symp. on Fault Tolerant Comp.*, pp. 98-105, 1983.
- [10] M. Sipser, "Introduction to the theory of Computation," *PWS Publishing Company*, 1997.
- [11] W. Kunz and P. Menon, "Multilevel Logic Optimization by Implication Analysis," *Proc. Int. Conj: Computer-Aided Design (ICCAD)*, 1994.
- [12] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *J. ACM*, vol. 7, pp. 201-215, 1960.
- [13] J. M. Silva and L. G. e Silva, "Solving satisfiability in combinational circuits with backtrack search and recursive learning", *XII Symposium on Integrated Circuits and Systems Design*, pp. 192-195, 2000.
- [14] J. M. Silva and L. G. e Silva, "Solving satisfiability in combinational circuits", *IEEE Design and Test of computers*, pp. 16-21, 2003.
- [15] W. Cao and D. K. Pradhan, "Sequential Redundancy Identification using recursive learning", *IEEE/ACM International Conference on Computer-Aided Design*, pp. 56-62, 1996.