Algorithms for Satisfiability in Combinational Circuits Based on Backtrack Search and Recursive Learning

João P. Marques-Silva and Luís Guerra e Silva

IST/INESC
Cadence European Laboratories
R. Alves Redol, 9, 1
1000 Lisboa, Portugal
{jpms,lgs}@algos.inesc.pt

Contact Author: João Marques-Silva, Phone 351-1-310-0387, Fax 351-1-314-5843, Email jpms@inesc.pt

Algorithms for Satisfiability in Combinational Circuits Based on Backtrack Search and Recursive Learning

Abstract

Boolean Satisfiability is a ubiquitous modeling tool in Electronic Design Automation (EDA). It finds application in test pattern generation, delay-fault testing, combinational equivalence checking and circuit delay computation, among many other problems. Moreover, Boolean Satisfiability is also in the core of algorithms for solving Binate Covering Problems. This paper starts by describing how Boolean Satisfiability algorithms can take circuit structure into account when solving instances derived from combinational circuits. Afterwards, it shows how recursive learning techniques can be incorporated into Boolean Satisfiability algorithms. The proposed algorithmic framework has several natural applications in EDA. Moreover, potential advantages include smaller run times, the utilization of circuit-specific search pruning techniques, avoiding the overspecification problem that characterizes Boolean Satisfiability testers, and reducing the time for iteratively generating instances of SAT from circuits. The experimental results obtained on a large number of benchmark examples in different problem domains display dramatic reductions in the run times of the algorithms, and provide clear evidence that computed solutions can have significantly less specified variable assignments than those obtained with common SAT algorithms.

1 Introduction

Boolean Satisfiability (SAT) is intrinsic to many problems in Electronic Design Automation (EDA). Originally motivated by the work of T. Larrabee in test pattern generation [22], SAT models and techniques have since been applied to delay-fault testing, combinational equivalence checking, circuit delay computation, logic synthesis and functional vector generation, among other applications. (See [7, 9, 10, 22, 24, 26, 29] for an overview of applications of SAT to EDA.) Moreover, SAT can also play a central role in solving instances of binate covering problems (BCP) [8, 11, 12, 14, 15], in particular for those in which the constraints are hard to satisfy, e.g. in computing minimum size test patterns [12]. SAT also plays a key role in other domains, including for example Artificial Intelligence [3, 30] and Operations Research [2]. Recent years have seen dramatic improvements in SAT algorithms, which have been thoroughly validated in different application areas [3, 23, 30].

With respect to applications of SAT in EDA, in most cases the original problem formulation starts from a circuit description, for which a given (circuit) property 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.

The utilization of CNF models and SAT algorithms has important advantages:

1. Existing, and extensively validated SAT algorithms, can be used instead of dedicated algorithms.

2. New improvements and new SAT algorithms can be easily applied to each target application.

In contrast, the utilization of CNF formulas and associated SAT algorithms is also characterized by several drawbacks:

1. As observed in [29], the structural information of the circuit, often of crucial importance, is lost.

2. In many EDA problems, a large number of instances of SAT has to be solved for each circuit. Hence, mapping a given problem description into SAT can represent a significant percentage of the overall running time [22].

3. Computed input patterns are in general overspecified. Overspecification can be a serious drawback in different applications, including circuit testing and binate constraint solving.

4. Powerful circuit-based reasoning techniques, e.g. recursive learning [19-21], cannot easily be applied.

With the purpose of addressing these problems, in [29] a new dynamic data structure, i.e. an extended implication graph, is proposed for solving instances of SAT in combinational circuits. Despite the promising results of [29], utilizing a new data structure requires dedicated algorithms. Hence new search pruning techniques, developed for example in the context of SAT algorithms, will have to be adapted to the circuit graph data structure.

In this paper we show how to utilize structural information in SAT algorithms. To a generic SAT algorithm we add
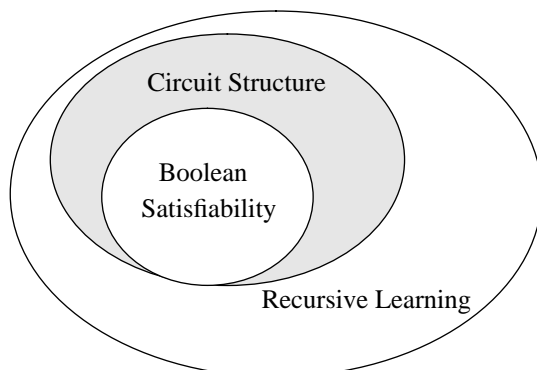
Figure 1: Integrating circuit structure and recursive learning into Boolean satisfiability algorithms
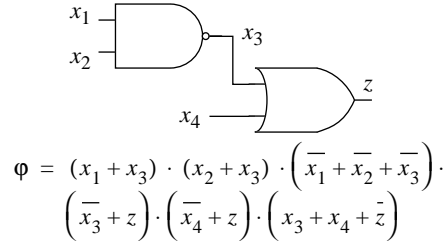
a layer that maintains circuit-related information, e.g. fanin/fanout information as well as value justification relations. The proposed approach allows using any SAT algorithm to which this layer can be added. The main advantages of the proposed approach is that some of the previously mentioned drawbacks, i.e. inaccessibility to structural information and overspecification of input patterns, are eliminated. The main contribution over the work of [29] is that data structures used for SAT need not be modified, and so existing algorithmic solutions for SAT can naturally be augmented with the proposed layer for handling structural information. Moreover, the approach proposed in this paper is significantly simpler than the one in [29], since only minor modifications to SAT algorithms are required.

Another contribution of this paper is describing how to extend recursive learning to solving Boolean Satisfiability. One practical consequence is that the resulting SAT algorithms can be competitive with existing solutions for solving equivalence checking [5, 17, 18, 20, 27, 29][1], whereas existing SAT algorithms are not. Another consequence is that recursive learning becomes applicable to other problem domains. We should also note that the recursive learning procedure proposed in this paper is strictly stronger than the original algorithm [21], since it learns and records *clauses*, in contrast with the original recursive learning procedure, which is only targeted at learning *necessary assignments*. If recursive learning is used in the context of search, the ability to record clauses can become a significant advantage. Moreover, we show how the proposed extended recursive learning procedure can take into account circuit structure information in order for reducing the amount of reasoning effort and consequently the CPU time.

The proposed approach for solving SAT in combinational circuits is graphically illustrated in Figure 1. Basically, by adding new layers to an existing SAT algorithm, we can exploit circuit structure and can incorporate search techniques specifically targeted for combinational circuits.

The paper is organized as follows. Section 2 introduces basic definitions associated with SAT and combinational circuits. Next we briefly survey SAT algorithms, giving particular emphasis to those that have been shown to be effective in solving EDA problems. Afterwards, in Section 4, we detail the proposed approach for taking into consideration structural information while solving SAT. Section 5 shows how to extend recursive learning to CNF formulas

---

1. See for example [16] for a survey of different approaches for solving combinational equivalence checking.

$$\varphi \; = \; (x_1 + x_3) \cdot (x_2 + x_3) \cdot \left(\overline{x_1} + \overline{x_2} + \overline{x_3} \right) \cdot$$
$$\left(\overline{x_3} + z \right) \cdot \left(\overline{x_4} + z \right) \cdot \left(x_3 + x_4 + \overline{z} \right)$$

(a) Consistent assignments

$$\varphi' \; = \; (x_1 + x_3) \cdot (x_2 + x_3) \cdot \left(\overline{x_1} + \overline{x_2} + \overline{x_3} \right) \cdot$$
$$\left(\overline{x_3} + z \right) \cdot \left(\overline{x_4} + z \right) \cdot \left(x_3 + x_4 + \overline{z} \right) \cdot \left(\overline{z} \right)$$

(b) With property $z = 0$

Figure 2: Example circuit and CNF formula

and how to then exploit the available circuit structure. Section 6 analyzes preliminary results on two EDA applications. The paper concludes in Section 7 by reviewing the contributions and providing some perspective on future research work.

2 Definitions

The CNF formula of a combinational circuit is the conjunction of the CNF formulas for each gate output, where the CNF formula of each gate denotes the valid input-output assignments to the gate. An example of a circuit, associated CNF formula and the specification of an objective is shown in Figure 2. (The derivation of the CNF formulas for simple gates can be found for example in [22].) If we view a CNF formula for a gate as a set of clauses, the CNF formula φ for the circuit is defined by the set union (or conjunction) of the CNF formulas of each gate. Hence, given a combinational circuit it is straightforward to create the CNF formula for the circuit as well as the CNF for proving a given property of the circuit.

SAT algorithms operate on CNF formulas, and consequently can readily be applied to solving instances of SAT associated with combinational circuits. Examples include the CNF formulas for test pattern generation [22] and circuit delay computation expressions [26].

3 Boolean Satisfiability Algorithms

The overall organization of a generic SAT algorithm is shown in Figure 3. This generic SAT algorithm captures the organization of several of the most competitive algorithms [3, 23, 30].

The algorithm conducts a search through the space of the possible assignments to the problem instance variables. At each stage of the search, a variable assignment is selected with the `Decide()` function. A decision level d is asso-

```
// Input arg:        Current decision level d
// Output arg:       Backtrack decision level β
// Return value:     SATISFIABLE or UNSATISFIABLE
//
SAT (d, &β)
{
    if (Decide (d) != DECISION)
        return SATISFIABLE;
    while (TRUE) {
        if (Deduce (d) != CONFLICT) {
          if (SAT (d + 1, β) == SATISFIABLE)
            return SATISFIABLE;
          else if (β != d || d == 0) {
             Erase (d); return UNSATISFIABLE;
          }
        }
        if (Diagnose (d, β) == CONFLICT) {
          return UNSATISFIABLE;
        }
    }
}
```

Figure 3: Generic backtrack search SAT algorithm

ciated with each selection of an assignment. Implied necessary assignments are identified with the `Deduce()` function, which in most cases corresponds to straightforward derivation of implications [3, 23]. Whenever a clause becomes unsatisfied the `Deduce()` function returns a conflict indication which is then analyzed using the `Diagnose()` function. The diagnosis of a given conflict returns a backtracking decision level β, which denotes the decision level to which the search process is required to backtrack to. The `Erase()` function clears implied assignments that result from each assignment selection. Different organizations of SAT algorithms can be modeled by this generic algorithm. Currently, all of the most efficient SAT algorithms [3, 23, 30] are characterized by several of the following key properties:

1. The analysis of conflicts can be used for implementing *Non-chronological Backtracking* search strategies. Hence, assignment selections deemed irrelevant can be skipped over during the search [3, 23, 30].

2. The analysis of conflicts can also be used for identifying and recording new implicates of the Boolean function associated with the CNF formula. *Clause Recording* plays a key role in recent SAT algorithms, but in most cases large recorded clauses are eventually deleted [3, 23].

3. Other techniques have been developed. *Relevance-Based Learning* [3] extends the life-span of large recorded clauses that will eventually be deleted. *Conflict-Induced Necessary Assignments* [23] denote assignments to variables which are necessary for preventing a given conflict from occurring again during the search.

Gate	$\upsilon_0(x)$	$\upsilon_1(x)$
$x = \text{AND}(w_1, ..., w_k)$	1	$\lvert FI(x) \rvert$
$x = \text{NAND}(w_1, ..., w_k)$	$\lvert FI(x) \rvert$	1
$x = \text{NOR}(w_1, ..., w_k)$	1	$\lvert FI(x) \rvert$
$x = \text{XOR}(w_1, ..., w_k)$	$\lvert FI(x) \rvert$	$\lvert FI(x) \rvert$

Table 1: Threshold values on assigned inputs

Before running the SAT algorithm, different forms of preprocessing can be applied [23]. This in general is denoted by a `Preprocess()` function.

4 Satisfiability in Combinational Circuits

4.1 Additional Data Structures

Let C_p denote a property of a combinational circuit C which is to be satisfied to an objective value o. This satisfiability problem is denoted by $\langle C_p, o \rangle$ and can be mapped into an instance of SAT, φ. The following information is associated with each variable x of φ, that also represents a circuit node x of C:

1. $FI(x)$ denotes the fanin nodes of x.

2. $FO(x)$ denotes the set of fanout nodes of x.

3. $\upsilon_v(x)$ denotes the threshold value on the number of suitable assigned inputs (of x) that are necessary for justifying value v on node x.

4. $\iota_v(x)$ denotes the actual counter of assigned inputs (of x) that are involved in justifying the value v on node x.

Note that the value assigned to each variable x is denoted by $\nu(x)$. Moreover, observe that each circuit node x, with assigned value v, becomes justified whenever $\iota_v(x) \geq \upsilon_v(x)$.

Table 1 contains a few examples of threshold values on the number of assigned inputs required for justifying a given node. For example, for an AND gate at least one input assigned value 0 justifies the assignment of value 0 to x, whereas for value 1 all inputs must be assigned value 1. Hence, $\upsilon_0(x) = 1$ and $\upsilon_1(x) = \lvert FI(x) \rvert$. As another example, observe that for an XOR gate justification of any assigned value requires assignments to all gate inputs; hence $\upsilon_0(x) = \upsilon_1(x) = \lvert FI(x) \rvert$. For other simple gates this information can also be easily derived, and in all cases we have $\upsilon_0(x), \upsilon_1(x) \in \{1, \lvert FI(x) \rvert\}$.

For any simple gate with output x, we can associate with each fanin node w the counters that must be updated as the result of assigning a value v to w. For example, for an AND gate an assignment of 0 to a fanin node w increments

Gate	$w_i = 0$	$w_i = 1$
$x = \text{AND}(w_1, ..., w_k)$	$\iota_0(x)$	$\iota_1(x)$
$x = \text{NAND}(w_1, ..., w_k)$	$\iota_1(x)$	$\iota_0(x)$
$x = \text{NOR}(w_1, ..., w_k)$	$\iota_1(x)$	$\iota_0(x)$
$x = \text{XOR}(w_1, ..., w_k)$	$\iota_0(x)$ $\iota_1(x)$	$\iota_0(x)$ $\iota_1(x)$

Table 2: Justification counters associated with gate inputs

$\iota_0(x)$ by 1, and an assignment of 1 to fanin node w increments $\iota_1(x)$ by 1. These relations are illustrated in Table 2 for a few example gates. Note that for the XOR gates, both counters are updated when an input node becomes assigned.

As with standard search algorithms in combinational circuits [1], a *justification frontier* is maintained, which denotes the sets of variables/nodes that require justification. Observe that the condition that indicates the need for node justification is $(\nu(x) = v) \wedge (\iota_v(x) < \upsilon_v(x))$, where $v \in \{0, 1\}$.

4.2 Modifications to the SAT Algorithm

Given the previous definitions, a SAT algorithm can be adapted so that the information regarding justification can be properly maintained. Moreover, the fanin information can be used for implementing structure-based heuristic decision making procedures, e.g. *simple or multiple backtracing* [1]. With respect to the algorithm of Figure 3, functions `Deduce()` and `Diagnose()` have to invoke dedicated procedures for updating node justification information. Additionally, the `Decide()` function now tests for satisfiability by checking for an empty justification frontier instead of checking whether all clauses are satisfied. These are the only required modifications to the general SAT algorithm. In addition, the `Decide()` function can optionally be modified to perform backtracing given the fanin information associated with each variable.

We should note that the data structures described above operate in much the same way as justification works in combinational circuits [1]. The main difference is that in our approach justification and value consistency are formally dissociated; value consistency is handled by the SAT algorithm and justification by the new added layer.

5 Implementing Recursive Learning in SAT Algorithms

In this section we describe how to extend recursive learning [21] for CNF formulas. We start by briefly reviewing the basic reasoning principle supporting recursive learning and illustrate how it can be applied in solving instances of SAT. We then describe the changes to the basic backtrack search SAT algorithm so that it incorporates recursive learning.
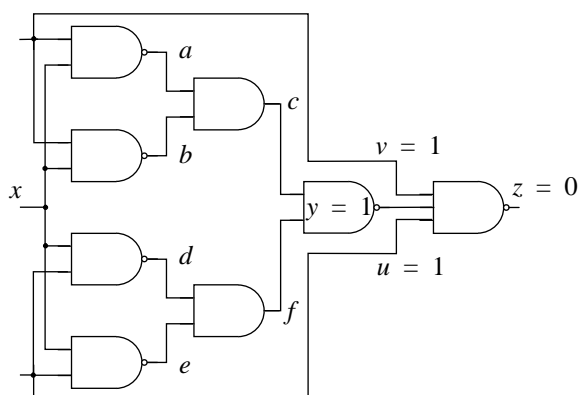
Figure 4: Example circuit

Assignments: $\{z = 1, u = 0\}$

$$\omega_1 = (u + x + \neg w)$$
$$\omega_2 = (x + \neg y)$$
$$\omega_3 = (w + y + \neg z)$$

Figure 5: Recursive learning on clauses

Let us consider the example circuit of Figure 4. Further, let us assume that our goal is to justify the objective $z = 0$. As a result, it is immediate to conclude that the assignments $(v = 1) \wedge (u = 1) \wedge (y = 1)$ are required. Assuming that v and u are primary inputs we only need to consistently justify the assignment to node y. In order to do this we resort to recursive learning with depth 2 [21].

For $y = 1$ the first justification to be considered is $c = 0$. The possible justifications for this assignment are either $a = 0$ or $b = 0$. Considering the justification $a = 0$, the assignment $x = 1$ is implied. The same implication results when considering the justification $b = 0$. Hence, we can conclude that the assignment $c = 0$ implies the assignment $x = 1$. From the initial recursion, the remaining justification for $y = 1$ is $f = 0$. The possible justifications for $f = 0$ are either $d = 0$ or $e = 0$. Considering each justification individually yields once more the assignment $x = 1$. Since this assignment is implied for any justification of $y = 1$, we can then conclude that the assignment $y = 1$ implies the assignment $x = 1$ if consistent assignments are to be identified for the circuit nodes.

The same reasoning that is used for implementing recursive learning in combinational circuits can naturally be extended to clauses in CNF formulas. Indeed, for any clause to be satisfied at least one of the yet unassigned literals *must* be assigned value 1. Recursive learning on CNF formulas consists of studying the different ways of satisfying a given selected clause and identifying common assignments, which are then deemed *necessary* for the clause to become satisfied and consequently for the instance of SAT to be satisfiable. Clearly, and because conflict diagnosis can also be implemented, each identified assignment needs to be adequately *explained*. Consequently, with each identified assignment a clause that describes *why* the assignment is necessary is created. Let us consider the example CNF formula of Figure 5. In order to satisfy clause ω_3, either $w = 1$ or $y = 1$. Considering each assignment separately leads to the implied assignment $x = 1$; for $w = 1$ due to ω_1 and for $y = 1$ due to ω_2. Hence, the assignment $x = 1$ is necessary if the CNF formula is to be satisfied. One sufficient explanation for this implied assignment is

given by the logical implication $(z = 1) \wedge (u = 0) \Rightarrow (x = 1)$, which can be represented in clausal form as $(\neg z + u + x)$. Consequently, this clause represents a new *implicate* of the Boolean function associated with the CNF formula and so it can be added to the CNF formula. This new clause also implies the assignment $x = 1$ as long as $z = 1$ and $u = 0$, as intended. As with recursive learning for combinational circuits, recursive learning for CNF formulas can be generalized to any recursion depth.

In backtrack search SAT algorithms, recursive learning can be implemented as part of the `Preprocess()` function or as part of the `Deduce()` function. First, during preprocessing, each variable is assigned both logic values and implied assignments are identified each time. Second, either during the search or as part of preprocessing, clauses with literals set to 0 are analyzed by evaluating the consequences of each assignment that satisfies the clause. Assignments common to all clause justifications are deemed necessary. This procedure is iteratively applied to clauses with literals assigned value 0 as a result of the most recent implication sequence. Finally, for either preprocessing or deduction, this process is repeated at each recursion depth. Observe that at each step, each identified necessary assignment is associated with a newly created clause, that corresponds to a sufficient *explanation* for the assignment to be necessary.

Observe that our proposed recursive learning procedure derives and *records* implicates of the function associated with the CNF formula. Clearly, these implicates prevent repeated derivation of the same assignments during the subsequent search. In contrast, the recursive learning procedure developed for combinational circuits only records necessary assignments [21]. Hence, when used as part of a search algorithm, recursive learning may eventually re-derive some of the already derived necessary assignments. Moreover, by taking into account the circuit structure information, the recursive learning procedure can be made simpler, since only clause justifications of nodes in the fanin of a given unjustified node need to be considered.

6 Experimental Results

In this section we evaluate the practical usefulness of the circuit structure-aware SAT algorithm described in Section 4. For this purpose, we used a state of the art public-domain SAT algorithm, GRASP [23], and built on top of this algorithm a new SAT algorithm that takes structural information into account, CGRASP. Moreover, the extended recursive learning procedure can also be utilized, either in GRASP or in CGRASP. In this situation we refer to the SAT algorithms as, respectively, RL_GRASP and RL_CGRASP.

Three EDA applications are considered for evaluating CGRASP, RL_GRASP and RL_CGRASP, namely test pattern generation [22], circuit delay computation (CDC) [9, 26] and combinational equivalence checking. Statistics for the ISCAS'85 benchmark circuits [6], regarding TPG and circuit delay computation, are shown in Table 3. #PI, #PO, #G, #F, #D, #R, LTP and Δ denote, respectively, the number of primary inputs, the number of primary outputs, the number of gates, the number of stuck-at faults, the number of detectable faults, the number of redundant faults, the longest topological delay and the critical circuit delay under floating-mode operation [9]. The miter instances for combinational equivalence checking are described in Section 6.3. All the experimental results shown below were obtained on a P-II 266 MHz Linux workstation with 128 MByte of physical memory.

Circuit	#PI	#PO	#G	TPG			CDC	
				#F	#D	#R	LTP	Δ
C432	36	7	160	524	520	4	17	17
C499	41	32	202	758	750	8	11	11
C880	60	26	383	942	942	0	24	24
C1355	41	32	546	1574	1566	8	24	24
C1908	33	25	880	1879	1870	9	40	37
C2670	233	140	1193	2747	2630	117	32	30
C3540	50	22	1669	3428	3291	137	47	46
C5315	178	123	2307	5350	5291	59	49	47
C6288	32	32	2406	7744	7710	34	124	123
C7552	207	108	3512	7550	7419	131	43	42

Table 3: Statistics for ISCAS'85 circuits

Circuit	CNF(s)	GRASP				CGRASP			
		#B	#NCB	%S	SAT(s)	#B	#NCB	%S	SAT(s)
C432	3.2	2,855	115	100	7.5	167	8	75	2.2
C499	5.1	1,254	1,110	100	13.0	24	24	82	3.0
C880	4.9	2,690	968	99	30.2	55	31	33	3.1
C1355	27.9	5,464	644	100	77.4	73	40	89	18.9
C1908	27.6	3,505	2,224	100	108.0	1,353	946	80	33.6
C2670	14.0	12,949	6,997	85	437.2	5,132	1296	26	34.2
C3540	66.4	33,049	12,568	99	839.0	692	199	66	83.6
C5315	32.8	5,081	3,138	98	2,728	1,671	631	22	80.2
C6288	362.0	149,374	5,360	100	4,215	16,314	398	77	467.2
C7552	65.2	70,958	35,322	98	12,641	4,811	1704	44	248.0

Table 4: Results for test pattern generation

6.1   Test Pattern Generation

The first experiment consists in evaluating CGRASP for TPG. For this purpose, both GRASP and CGRASP were run on the ISCAS'85 benchmark circuits. The results are shown in Table 4. For each algorithm, CNF(s), #B, #NCB, %S and SAT(s) denote, respectively the total CNF formula build time, the total number of backtracks, the total number of non-chronological backtracks, the average percentage of specified variable assignments over all solved instances and the SAT search time. As can be readily concluded, the search times become drastically reduced when structural information is taken into account, and a justification frontier is used for SAT purposes. Indeed, for some of the benchmark circuits the search times can be reduced by almost two orders of magnitude.

Besides the reduction in CPU times from GRASP to CGRASP, we can also observe similar reductions in the total number of backtracks. Nevertheless, the pruning techniques of GRASP are still used in CGRASP, as the number of

Circuit	#PI	#PO	#G	LTP	Δ
csa.32.16	65	33	170	69	66
csa.32.8	65	33	180	73	38
csa.32.4	65	33	200	81	30
csa.64.16	129	65	340	137	70
csa.64.8	129	65	360	145	46
csa.64.4	129	65	400	161	46
csa.128.16	257	129	680	273	78
csa.128.8	257	129	720	289	62
csa.128.4	257	129	800	321	78

Table 5: Statistics for carry-skip adders

non-chronological backtracks clearly illustrates. Moreover, CGRASP computes test patterns that are significantly less specified than the test patterns computed by GRASP (see Table 4). Hence, by taking structural information into account we can effectively handle the overspecification problem, and compute test patterns that can be as specified as those obtained with purely structural methods [1].

6.2 Circuit Delay Computation

Another potential application is SAT-based circuit delay computation [26]. Experimental results comparing GRASP and CGRASP for circuit delay computation are shown in Table 6. For these experiments, the model of [26] is used, and unit gate delays are assumed. In addition to the ISCAS'85 circuits, we generated several carry-skip adders[2], to evaluate how each algorithm performs with increasing circuit size and complexity. As can be concluded once more, utilizing to the structural information of the circuit proves crucial in reducing the CPU times spent searching. Similarly to TPG, the reduction in run times can be dramatic, in some cases two orders of magnitude reductions are observed. Once more, despite the very significant reductions in the run times, we can still observe the pruning techniques of GRASP being used. As shown, the number of non-chronological backtracks still represents a significant percentage of the overall number of backtracks. Moreover, as was noted for TPG, the advantages of CGRASP become patent with increasing circuit sizes, especially for the carry-skip adders [26].

6.3 Combinational Equivalence Checking

In this section we study the application of existing SAT algorithms for solving combinational equivalence checking (CEC). In addition we utilize two SAT algorithms that incorporate recursive learning and are aware of circuit structure. The first RL_GRASP, utilizes recursive learning but does not exploit circuit structure information. The second one, RL_CGRASP, utilizes both recursive learning and the circuit structure information to reduce the number of clauses that have to be analyzed during the application of recursive learning. Both algorithms were run with the following options for all benchmark instances:

---

2. See Table 5 for statistics regarding these circuits.

Circuit	CNF(s)	GRASP			CGRASP		
		#B	#NCB	SAT(s)	#B	#NCB	SAT(s)
C432	0.0	66	3	0.1	0	0	0.0
C499	0.0	0	0	0.5	0	0	0.0
C880	0.0	20	12	0.3	0	0	0.0
C1355	0.1	1,540	397	9.0	1	1	0.1
C1908	0.1	93	67	9.3	56	42	1.3
C2670	0.1	558	231	21.9	422	177	2.2
C3540	0.1	41	30	3.9	3	3	0.5
C5315	0.1	35	28	9.0	17	2	0.2
C6288	0.1	1,216	223	37.3	1,725	208	34.0
C7552	0.1	8	8	23.8	1	1	0.5
csa.32.16	0.0	0	0	0.2	0	0	0.0
csa.32.8	0.2	44	19	1.4	0	0	0.2
csa.32.4	1.3	431	240	7.4	210	33	0.8
csa.64.16	1.2	104	47	10.7	0	0	1.0
csa.64.8	6.0	1,745	425	80.0	694	57	4.7
csa.64.4	17.8	7,994	3,058	414.8	3,510	1,713	19.5
csa.128.16	36.2	15,110	2,500	1,447	2,526	105	31.4
csa.128.8	89.5	56,570	15,122	5,853	11,053	3,665	93.2
csa.128.4	199.2	108,098	44,309	24,725	33,190	23,238	399.6

Table 6: Results for circuit delay computation

1. Preprocess the CNF formula using depth 1 recursive learning for CNF formulas.

2. Search for a solution using recursive learning of depth 1 in Deduce() (See Figure 3), i.e. apply recursive learning at *each* level in the decision tree. Furthermore, clauses of size no greater than 80 can be recorded.

In order to evaluate the different SAT algorithms the ISCAS'85 miters [20] are used. The results are shown in Table 7. For each algorithm and for each instance, the allowed CPU time was 3,000 seconds. A first conclusion is that the most efficient SAT algorithms, including REL_SAT [3], SATO [30] and GRASP [23] are in general inadequate for solving instances of CEC. In contrast, by including recursive learning, RL_GRASP and RL_CGRASP are able to solve *all* instances in reasonable amounts of CPU time. RL_GRASP can in general be faster, but it also experiences larger variations in the run times, hence being less robust. Another interesting result is that the other features of efficient SAT algorithms, including non-chronological backtracking, actually occur while solving instances of CEC, for both RL_GRASP and RL_CGRASP. As can be observed, the number of non-chronological backtracks (#NCB) can be a significant percentage of the overall number of backtracks (#B). Moreover, the value of the *largest backjump* in the decision tree (LJ) can be significant, thus justifying using conflict diagnosis techniques in combinational equivalence checking.

Class	Circuit	REL-SAT	SATO	GRASP	RL_GRASP				RL_CGRASP			
					time	#B	#NCB	LJ	time	#B	#NCB	LJ
Standard miters (unsatisfiable)	C432	1.4	11.7	2.1	1.4	8	1	2	1.8	1	0	1
	C499	19.5	> 3,000	> 176.0	3.4	0	—	—	4.4	0	—	—
	C1355	> 3,000	> 3,000	> 3,000	9.0	0	—	—	13.7	0	—	—
	C1908	> 3,000	> 3,000	394.2	47.4	5	1	4	76.5	7	1	4
	C2670	> 3,000	> 3,000	991.9	28.2	19	13	8	37.3	14	10	23
	C3540	> 3,000	631.3	> 3,000	2,003	3,727	961	22	1,280	1,916	568	14
	C5315	> 3,000	> 3,000	> 493.8	222.7	618	352	109	238.3	505	248	64
	C6288	> 3,000	> 3,000	> 346.4	54.8	0	—	—	21.8	0	—	—
	C7552	> 3,000	> 3,000	> 2,400	1,062	592	290	62	1,434	321	94	66
Incorrect miters (satisfiable)	C1908	> 3,000	0.34	258.9	47.5	0	—	—	78.4	7	2	3
	C2670	0.2	> 3,000	11.9	29.3	0	—	—	38.8	0	—	—
	C3540	24.8	N/A[a]	> 2,013	317.8	761	205	10	464.9	682	147	7
	C5315	> 3,000	> 3,000	35.5	135.1	413	210	35	155.7	203	63	62
	C7552	2,409	> 3,000	95.7	735.6	0	—	—	1,158	0	—	—

Table 7: Results for the ISCAS miters using SAT algorithms

a. SATO gives an incorrect result for this instance.

7 Conclusions

This paper proposes a new algorithm for solving Boolean Satisfiability problems in combinational circuits. For manipulating structural information, the proposed approach requires minor modifications in existing SAT algorithms. Moreover, we illustrate how recursive learning can also be used for solving SAT and, when integrated in a SAT algorithm, how it can exploit structural information for simplifying the amount of reasoning performed. The experimental evaluation of the new algorithm on different applications shows dramatic improvements over other Boolean Satisfiability solvers that have previously been shown to be highly effective [23]. The experimental results also indicate that the proposed algorithm effectively addresses the overspecification problem, leading to significant reductions in the number of assigned variables for the satisfiable instances of SAT.

References

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

[2] P. Barth, "A Davis-Putnam Enumeration Algorithm for Linear pseudo-Boolean Optimization," Technical Report MPI-I-95-2-003, Max Planck Institute for Computer Science, 1995.

[3] R. Bayardo Jr. and R. Schrag, "Using CSP Look-Back Techniques to Solve Real-World SAT Instances," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 203-208, July 1997.

[4] C. L. Berman and L. H. Trevillyan, "Functional Comparison of Logic Designs for VLSI Circuits," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 456-459, November 1989.

[5] D. Brand, "Verification of Large Synthesized Designs," in *Proceedings of International Conference on Computer-Aided*

Design, pp. 534-537, November 1993.

[6]   F. Brglez and H. Fujiwara, "A Neutral List of 10 Combinational Benchmark Circuits and a Target Translator in FORTRAN," in *Proceedings of the International Symposium on Circuits and Systems*, 1985.

[7]   C.-A. Chen and S. K. Gupta, "A Satisfiability-Based Test Generator for Path Delay Faults in Combinational Circuits," in *Proceedings of the Design Automation Conference*, pp. 209-214, June 1996.

[8]   O. Coudert, "On Solving Covering Problems," in *Proceedings of the Design Automation Conference*, June 1996.

[9]   S. Devadas, K. Keutzer and S. Malik, "Computation of Floating Mode Delay in Combinational Circuits: Practice and Implementation," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 12 no. 12, pp. 1923-1936, December 1993.

[10] F. Fallah, S. Devadas and K. Keutzer, "Functional Vector Generation For HDL Models Using Linear Programming and 3-Satisfiability", in *Proceedings of the Design Automation Conference*, pp. 528-533, June 1998.

[11] F. Ferrandi et al., "Symbolic Algorithms for Layout-Oriented Synthesis of Pass Transistor Logic Circuits," to appear in *Proceedings of the International Conference on Computer-Aided Design*, November 1998.

[12] P. Flores, H. Neto and J. Marques-Silva, "An Exact Solution to the Minimum-Size Test Pattern Problem" to appear in *Proceedings of the International Conference on Computer Design,* October 1998.

[13] J. W. Freeman, *Improvements to Propositional Satisfiability Search Algorithms*, Ph.D. Dissertation, Department of Computer and Information Science, University of Pennsylvania, May 1995.

[14] R. Fuhrer and S. Nowick, "Exact Optimal State Minimization for 2-Level Output Logic," in *International Workshop on Logic Synthesis*, June 1998.

[15] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*, Kluwer Academic Publishers, 1996.

[16] S.-Y. Huang and K.-T. Cheng, *Formal Equivalence Checking and Design Debugging*, Kluwer Academic Publishers, 1998.

[17] J. Jain, R. Mukherjee and M. Fujita, "Advanced Learning Techniques Based on Learning," in *Proceedings of the Design Automation Conference*, pp. 420-426, June 1995.

[18] A. Kuehlmann and F. Krohm, "Equivalence Checking Using Cuts and Heaps," in *Proceedings of the Design Automation Conference*, pp. 263-268, June 1997.

[19] W. Kunz and D. K. Pradhan, "Recursive Learning: An Attractive Alternative to the Decision Tree for Test Generation in Digital Circuits," in *Proceedings of the International Test Conference*, pp. 816-825, 1992.

[20] W. Kunz, "HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 538-543, November 1993.

[21] W. Kunz and D. Stoffel, *Reasoning in Boolean Networks*, Kluwer Academic Publishers, 1997.

[22] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *IEEE Transactions on Computer-Aided Design*, vol. 11, no. 1, pp. 4-15, January 1992.

[23] J. Marques-Silva and K. A. Sakallah, "GRASP—A New Search Algorithm for Satisfiability," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 220-227, November 1996. (URL: http://algos.inesc.pt/pub/users/jpms/ soft/grasp/grasp.tar.gz.)

[24] J. Marques-Silva and K. A. Sakallah, "Robust Search Algorithms for Test Pattern Generation," in *Proceedings of The International Symposium on Fault-Tolerant Computing*, pp. 152-161, June 1997.

[25] Y. Matsunaga, "An Efficient Equivalence Checker for Combinational Circuits," in *Proceedings of the Design Automation Conference*, June 1996.

[26] P. McGeer, A. Saldanha, P. R. Stephan, R. K. Brayton and A. L. Sangiovanni-Vincentelli, "Timing Analysis and Delay-Test Generation Using Path Recursive Functions," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 180-183, November 1991.

[27] S. Reddy, W. Kunz and D. Pradhan, "Novel Verification Framework Combining Structural and OBDD Methods in a Synthesis Environment," in *Proceedings of the Design Automation Conference*, pp. 414-419, June 1995.

[28] P. Stephan, R.K. Brayton and A.L. Sangiovanni-Vincentelli, "Combinatorial Test Generation Using Satisfiability", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 15, no. 9, Sep. 1996.

[29] P. Tafertshofer, A. Ganz and M. Henftling, "A SAT-Based Implication Engine for Efficient ATPG, Equivalence Checking, and Optimization of Netlists," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 648-657, November 1997.

[30] H. Zhang, "SATO: An Efficient Propositional Prover," in *Proceedings of International Conference on Automated Deduction*, pp. 272-275, July 1997.