Combinational Equivalence Checking Using Satisfiability and Recursive Learning

João Marques-Silva
Instituto Superior Técnico
Cadence European Labs/INESC
1000 Lisboa, Portugal
e-mail: jpms@inesc.pt

Thomas Glass
Siemens AG
Corporate Technology
81730 Munich, Germany
e-mail: thomas.glass@mchp.siemens.de

Abstract

The problem of checking the equivalence of combinational circuits is of key significance in the verification of digital circuits. In recent years, several approaches have been proposed for solving this problem. Still, the hardness of the problem and the ever-growing complexity of logic circuits motivates studying and developing alternative solutions. In this paper we study the application of Boolean Satisfiability (SAT) algorithms for solving the Combinational Equivalence Checking (CEC) problem. Although existing SAT algorithms are in general ineffective for solving CEC, in this paper we show how to improve SAT algorithms by extending and applying Recursive Learning techniques to the analysis of instances of SAT. This in turn provides a new alternative and competitive approach for solving CEC. Preliminary experimental results indicate that the proposed improved SAT algorithm can be useful for a large variety of instances of CEC, in particular when compared with pure BDD-based approaches.

1. Introduction

The problem of checking the equivalence of combinational circuits is of key significance in the verification of digital circuits, and has been the subject of significant contributions in recent years. As a result, several approaches have been proposed for solving the *Combinational Equivalence Checking* (CEC) problem. These approaches can be characterized as being structure-based [3, 9, 20], function-based [12], or a mix of the two [6, 8, 17][1]. Structure-based approaches can either use ATPG algorithms [3], or recursive learning [9, 10], whereas mixed approaches use structural information, Reduced Ordered Binary Decision Diagrams (ROBDDs) and, in some cases, different forms of learning [6, 17]. Approaches that relate and substitute internal circuit nodes can exhibit the *false negative* problem [2], i.e. declaring two circuits not equivalent when they are in fact equivalent. Different techniques have been proposed for handling the false negative problem [9, 13]. Despite the recent improvements for solving the CEC problem, its computational hardness and the ever-growing complexity of logic circuits motivates studying, developing and evaluating new alternative algorithmic solutions.

In this paper we study the application of Boolean Satisfiability (SAT) algorithms to equivalence checking. As will be shown below, even the most efficient SAT algorithms can in general be inadequate for CEC. Hence, we proposed to improve existing SAT algorithms with new techniques, that are suitable for solving combinational equivalence checking, and which may also be applicable to other problem domains. In particular, in this paper we describe how to extend recursive learning to solving Boolean Satisfiability. One practical consequence is that the resulting SAT algorithms are competitive for equivalence checking. 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 [10], 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.

Besides describing the extended recursive learning procedure, we detail its integration into an existing SAT algorithm [13], which includes other effective pruning techniques, and experimentally validate this new algorithm on a large number of combinational equivalence checking instances.

The paper is organized as follows. Section 2 introduces a few definitions used throughout the paper. Afterwards, we briefly review the organization of a SAT algorithm and, in Section 4, we show how to extend recursive learning to CNF formulas. Section 5 briefly describes different combinational equivalence checking strategies. Next, in Section 6 we provide experimental evidence supporting the utilization of SAT algorithms in CEC. Section 7 concludes the paper.

2. Preliminaries

The Conjunctive Normal Form (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. (The derivation of the CNF formulas for simple gates can be found for example in [11].) If we view a CNF formula as a set of clauses, the CNF formula φ for the circuit is defined by the set union (or the conjunction) of the CNF formulas for each gate. Hence, given a combinational circuit it is straightforward to create the CNF for-

1. Huang and Cheng [5] provide an alternative taxonomy, that distinguishes between incremental and symbolic approaches.

```
// 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 1: Generic backtrack search SAT algorithm

mula for the circuit as well as the CNF formula for proving propositional properties of the circuit [11].

SAT algorithms operate on CNF formulas, and consequently can readily be applied to solving instances of SAT associated with combinational circuits. For example, the *miter*[2] [3] structure can be readily mapped into a CNF formula and be solved with any SAT package.

3. Boolean Satisfiability Algorithms

Boolean Satisfiability algorithms find many applications in Electronic Design Automation, that include test pattern generation [11, 19], delay-fault testing and circuit delay computation. The recent utilization of SAT algorithms for solving different problems in EDA has been mostly motivated by the work of T. Larrabee on circuit testing [11]. Besides the above applications, another potential application of SAT is combinational equivalence checking [20]. The overall organization of a generic SAT algorithm is shown in Figure 1. This SAT algorithm captures the organization of several of the most competitive algorithms [1, 4, 13, 19, 21]. (See [13] for a more detailed description of the organization of a SAT algorithm.)

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 associated 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 [13]. 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 assigned variables 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 implement a number of the following key properties:

1. The analysis of conflicts can be used for implementing *Non-Chronological Backtracking* search strategies. Hence, assignment selections that are deemed irrelevant can be skipped during the search [1, 13, 21].
2. The analysis of conflicts can also be used for identifying and recording new clauses that denote 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 [1, 13].
3. Other techniques have been developed. *Relevance-Based Learning* [1] extends the life-span of large recorded clauses that will eventually be deleted. *Conflict-Induced Necessary Assignments* [13] denote assignments of variables which are necessary for preventing a given conflict from occurring again during the search.

Before running the SAT algorithm, different forms of preprocessing can be applied [13]. This in general is denoted by a `Preprocess()` function that is executed before invoking the search process.

2. Given two copies of a combinational circuit, a *miter* is defined as the OR of the XOR of each pair of primary outputs, and where the set of primary inputs of the two circuits are the same.

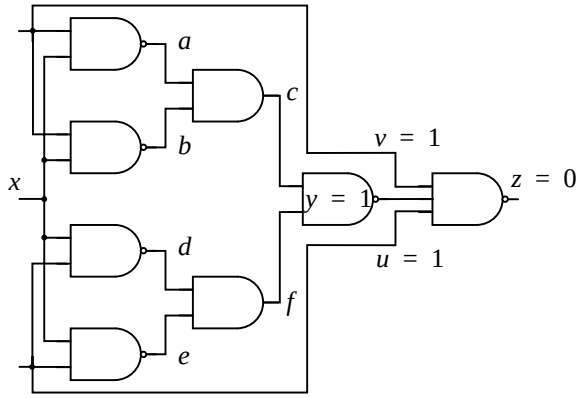

Figure 2: Example circuit

4. The Recursive Learning Procedure

In this section we describe how to extend recursive learning [10] 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.

Let us consider the example circuit of Figure 2. 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 [10].

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

$\omega_1 = (u + x + \neg w)$

$\omega_2 = (x + \neg y)$

$\omega_3 = (w + y + \neg z)$

Figure 3: Recursive learning on clauses

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 3. 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 [10]. Hence, when used as part of a search algorithm, the recursive learning procedure of [10] may eventually re-derive some of the already derived necessary assignments.

A more detailed description of utilizing recursive learning within SAT algorithms, as well as preliminary experimental results for instances of SAT, can be found in [14].

5. Equivalence Checking Framework

Different SAT-based approaches can be envisioned, either as complete algorithms or as a component of an incremental strategy for solving CEC:

1. First, create a miter for the circuit, and map the circuit into a CNF formula. Then invoke the SAT algorithm, trying to satisfy the (single) primary output to 1.
2. Run a BDD-based algorithm with a limit on the amount of allowed memory. Afterwards, run the SAT algorithm for each instance the BDD-based algorithm was unable to conclude.

As described in [5, 17] other approaches could also be used. We have implemented the above two approaches. The BDD-based algorithm followed by the SAT algorithm is by far the most competitive, even when a general-purpose non-optimized BDD package is used. Nevertheless, and interestingly, the SAT algorithm is by itself able to solve a very large set of practical instances of CEC. Even more interestingly, we provide strong evidence that by being restrictive in the amount of memory used by the BDD algorithm, one can get better CPU times.

We have implemented a new SAT algorithm, GRASP_RL, built on top of a publicly available SAT algorithm, GRASP [13]. For the results presented in this paper, GRASP-RL, is organized as follows:

1. Preprocess the CNF formula using depth 1 recursive learning for CNF formulas.
2. Search for a solution. Clauses of size no greater than 80 can be recorded.

Unless otherwise stated, the recursive learning is only applied as a preprocessing step, since it can be too time-consuming during the search to be executed at each decision step. After preprocessing, GRASP is run. Neverthe-

Class	Circuit	rel_sat	sato	grasp	grasp-rl			
					time	#B	#NCB	LJ
Standard miters (unsatisfiable)	C432	1.4	11.7	2.1	1.4	8	1	2
	C499	19.5	> 2,500	> 176.0	3.4	0	—	—
	C1355	> 2,500	> 2,500	> 2,500	9.0	0	—	—
	C1908	> 2,500	> 2,500	394.2	47.4	5	1	4
	C2670	> 2,500	> 2,500	991.9	28.2	19	13	8
	C3540	> 2,500	631.3	> 2,500	2,003	3,727	961	22
	C5315	> 2,500	> 2,500	> 493.8	222.7	618	352	109
	C6288	> 2,500	> 2,500	> 346.4	54.8	0	—	—
	C7552	> 2,500	> 2,500	> 2,400	1,062	592	290	62
Incorrect miters (satisfiable)	C1908	> 2,500	0.34	258.9	47.5	0	—	—
	C2670	0.2	> 2,500	11.9	29.3	0	—	—
	C3540	24.8	N/A[a]	> 2,013	317.8	761	205	10
	C5315	> 2,500	> 2,500	35.5	135.1	413	210	35
	C7552	2,409	> 2,500	95.7	735.6	0	—	—

Table 1: Results for the ISCAS miters using SAT algorithms

a. SATO [21] gives an error for this instance.

less, we note that in GRASP-RL recursive learning can be applied at *each* level of the decision tree.

6. Experimental Results

In this section, different algorithms for solving CEC are evaluated. Given the number of tools compared, different architectures were used. The CPU times presented correspond to approximately the equivalent times on a PII 266 MHz Linux machine with 128 MByte of physical memory. Some experiments were conducted on a SUN Ultra 170 workstation with 384 MByte of physical memory, and the run times were scaled accordingly.

In order to evaluate the different SAT algorithms, we start by analyzing the ISCAS'85 miters [9]. The results are shown in Table 1. For each algorithm and for each instance, the allowed CPU time was 2,500 seconds[3]. Moreover, in this experiment GRASP-RL was run with recursive learning of depth 1 at each level in the decision tree. A first conclusion is that the most efficient SAT algorithms, including REL_SAT [1], SATO [21] and GRASP [13] are in general inadequate for solving instances of CEC. In contrast, by including recursive learning, GRASP-RL is able to solve *all* instances in reasonable amounts of CPU times. Another interesting result, is that the other features of efficient SAT algorithms, including non-chronological backtracking, actually occur while solving instances of CEC. 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.

3. Instances that abort and terminate with a CPU time less than 2,500 seconds result from other computational resources being exceeded, in most cases the allowed number of recorded clauses.

# of circuits	Avg # inputs	Avg # outputs	Avg # gates
1,006	119.4	1	2,113.6

Table 2: Statistics for industry CEC benchmarks

memory	sis-bdd		tue-bdd		cudd	
	time	#aborted	time	#aborted	time	#aborted
16 MByte	6,372	547	13,853	296	141,909	28
32 MByte	8,297	89	35,491	290	154,787	5
64 MByte	10,352	89	58,145	202	147,407	5
128 MByte	14,993	89	53,465	197	162,492	5
256 MByte	19,388	89	66,790	195	153,544	5

Table 3: Results for BDD packages (time in seconds)

SAT algorithm	posit	rel_sat	sato	grasp	grasp-rl
# SAT	253	281	281	281	281
# UNSAT	32	49	501	604	725
# Aborted	721	676	224	121	0
Time (sec)	1,455,807	1,353,069	713,126	585,115	95,246

Table 4: Results for SAT algorithms

Besides the ISCAS'85 benchmarks, we also experimented with a large set of industrial equivalence checking instances[4]. Statistical data regarding these instances of CEC is shown in Table 2. Observe that each instance already denotes a miter [3]; hence the number of primary outputs is necessarily 1.

6.1. Results with BDD Packages

For the industrial instances, the results of using plain BDD packages with different amounts of allowed physical memory are shown in Table 3. For this experiment we utilized the BDD packages from SIS [18] (the CMU package), from T. U. Eindhoven (TUE_BDD) [7] and from University of Colorado (CUDD) [16]. For SIS and TUE_BDD, the default options were used. For the SIS BDD package, almost 10% of the instances are aborted even for large amounts of allowed memory. This in turn leads to increasing running times, with no observable improvements in the number of solved instances. Moreover, the TUE_BDD package is slower and requires more memory than SIS for these instances. For the CUDD package, dynamic ordering based on sifting was chosen, since for the default static ordering *all* instances were aborted (with 64 MByte of memory). With dynamic ordering this package becomes significantly slower, even though it is now able to solve all instances but five, provided the allowed memory is greater than or equal to 32 MByte.

6.2. Results with SAT Algorithms

We also ran several of the most efficient SAT algorithms on the industrial CEC instances. The obtained experimental results are shown in Table 4. For each benchmark the maximum allowed CPU time was 2,000 CPU

4. These instances have been kindly provided by Siemens AG, and represent parts of ASIC designs.

memory	16 Mbyte	32 Mbyte	64 Mbyte	128 Mbyte	256 Mbyte
sis+grasp-rl	51,483	8,418	10,473	15,114	19,509

Table 5: CPU times for SIS+GRASP-RL (in seconds)

seconds, and the allowed memory was 64 MByte. Instances not aborted can either be satisfiable (i.e. incorrect designs) or unsatisfiable (i.e. correct designs). From the results we can conclude, as before for the ISCAS'85 miters, that state-of-the-art SAT algorithms, including POSIT [4], REL_SAT [1], SATO [21] and GRASP [13] are in general inadequate for equivalence checking. In contrast, GRASP-RL, that applies recursive learning of depth 1 as a preprocessing step, is able to solve *all* problem instances in a reasonable amount of time. Note that even though GRASP-RL takes an order of magnitude more time than the SIS BDD package, it solves every instance whereas the SIS BDD package quits on 89 instances. In contrast, the run times of GRASP-RL are better than those of the CUDD package, that aborts on one instance. Besides GRASP-RL, of the other SAT algorithms, the most promising are the original version of GRASP and SATO. We note, however, that SATO [21] implements the same techniques that are used in GRASP.

6.3. Memory-Limited CEC

Another experiment is to run a BDD package with a limited amount of physical memory, and then run a SAT algorithm on the aborted instances. The results for this experiment, for different amounts of allowed memory, are shown in Table 5. A first observation is that the ideal solution may not be to run a BDD package with a large amount of allowed memory. Indeed, our results indicate that smaller run times can be obtained by reducing the amount of allowed memory while running the BDD package and then running a SAT package on the aborted instances. In our experiments, by allowing 32 MByte while running the SIS BDD package, we are able to solve each instance on average on 8.4 CPU seconds, in contrast with an average of more than 17 CPU seconds required by the more robust CUDD BDD package. This difference justifies utilizing the SIS+GRASP_RL approach as part of incremental structure-based CEC procedures [5].

7. Conclusions

In this paper we address the problem of solving combinational equivalence checking using Boolean Satisfiability algorithms. For this purpose, a new dedicated SAT algorithm that incorporates an extended recursive learning procedure for CNF formulas is described and has been implemented. Preliminary experimental results clearly indicate that SAT algorithms may be of practical use for CEC, either as stand-alone tools or as part of an incremental strategy for equivalence checking [5]. Moreover, the new SAT algorithm allows applying the recursive learning technique to any problem domain where SAT can also be used.

Additional work involves studying the utilization of other BDD-based algorithms for CEC [6, 8, 15] with SAT-based approaches, targeting minimum CPU times and reduced memory requirements. Moreover, evaluating larger recursion depths for the recursive learning procedure may provide useful insights.

References

[1] R. Bayardo Jr. and R. Schrag, "Using CSP Look-Back Techniques to Solve Real-World SAT Instances," in *Proc. of the Nat'l Conf. on Artificial Intelligence*, pp. 203-208, July 1997.

[2] C. L. Berman and L. H. Trevillyan, "Functional Comparison of Logic Designs for VLSI Circuits," in *Proc. of the Int'l Conf. on Computer-Aided Design*, pp. 456-459, November 1989.

[3] D. Brand, "Verification of Large Synthesized Designs," in *Proc. of the Int'l Conf. on Computer-Aided Design*, pp. 534-537, November 1993.

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

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

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

[7] G. Janssen, The Eindhoven BDD Package, Eindhoven University of Technology. (URL: ftp://ftp.ics.ele.tue.nl/pub/users/geert/bdd.tar.gz.)

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

[9] W. Kunz, "HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning," in *Proc. of the Int'l Conf. on Computer-Aided Design*, pp. 538-543, November 1993.

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

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

[12] S. Malik, A. R. Wang, R. K. Brayton and A. Sangiovanni-Vincentelli, "Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment", in *Proc. of the Int'l Conf. e on Computer-Aided Design*, pp. 6-9, November 1988.

[13] J. Marques-Silva and K. A. Sakallah, "GRASP—A New Search Algorithm for Satisfiability," in *Proc. of the Int'l Conf. on Computer-Aided Design*, pp. 220-227, November 1996. (URL: http://algos.inesc.pt/grasp/grasp.tar.gz.)

[14] J. Marques-Silva, "Improving Satisfiability Algorithms by Using Recursive Learning," in *International Workshop on Boolean Problems*, September 1998.

[15] Y. Matsunaga, "An Efficient Equivalence Checker for Combinational Circuits," in *Proc. of the Design Automation Conf.*, June 1996.

[16] S. Panda, F. Somenzi, and B. F. Plessier, "Symmetry detection and dynamic variable ordering of decision diagrams," in *Proc. of the Int'l Conf. on Computer-Aided Design*, pages 628-631, November 1994. (URL: ftp://vlsi.colorado.edu/pub/cudd-2.2.0.tar.gz.)

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

[18] E. Sentovich et al., "Sequential Circuit Design Using Synthesis and Optimization," in *Proc. of the Int'l Conf. on Computer Design, VLSI in Computers and Processors*, pp. 328-335, October 1992.

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

[20] P. Tafertshofer, A. Ganz and M. Henftling, "A SAT-Based Implication Engine for Efficient ATPG, Equivalence Checking, and Optimization of Netlists," in *Proc. of the Int'l Conf. on Computer-Aided Design*, pp. 648-657, November 1997.

[21] H. Zhang, "SATO: An Efficient Propositional Prover," in *Proc. of Int'l Conf. on Automated Deduction*, pp. 272-275, July 1997.