Preprint from Proceedings of
12th Asia and South Pacific Design Automation Conference, January 2007, Yokohama, Japan

Combinational Equivalence Checking Using Incremental SAT Solving, Output Ordering, and Resets

Stefan Disch Christoph Scholl

Albert-Ludwigs-University Freiburg
Institute of Computer Science
Georges-Köhler-Allee 51, 79110 Freiburg, Germany
e-mail: {disch, scholl}@informatik.uni-freiburg.de

Abstract— Combinational equivalence checking is an essential task in circuit design. In this paper we focus on SAT based equivalence checking making use of incremental SAT techniques which are well known from their application in Bounded Model Checking. Based on an analysis of shared circuit structures we present heuristics which try to maximize the benefit from incremental SAT solving in this application by looking for good orders in which the equivalence of different circuit outputs is checked. Moreover, we present a reset strategy for situations where the benefit from the incremental SAT approach seems to decrease. Experimental results demonstrate that our novel method outperforms traditional methods significantly.

I. INTRODUCTION

Checking whether a circuit implementation fulfills its specification, is a crucial task in VLSI CAD. During the last few years significant progress could be observed in this area, including advances in property checking, state space traversal and combinational equivalence checking. Many of these advances were based on the efficient symbolic representations of Boolean functions [1]; and on improvements to DPLL based satisfiability solvers – both concerning fundamental algorithmic methods and their efficient implementation [2–4].

For the purpose of this paper, combinational equivalence checking is of particular interest. Here, the task is to check whether the Boolean functions corresponding to the specification and the implementation are the same. Excluding functional validation by the application of test patterns, two main approaches can be used to perform the equivalence check: One possibility is the transformation of both implementation and specification into canonical forms like BDDs [1]. Then the equivalence check reduces to a comparison whether the canonical representations of implementation and specification are the same. Since BDD based methods may (depending on the benchmarks used) suffer from a high need for memory resources, many researchers have looked into an alternative method during the last years: Here the implementation and the specification are translated into one Boolean formula which is satisfiable if and only if the implementation and the specification do not realize the same Boolean function [5–7]. Modern SAT solvers may be used to solve the satisfiability problem [3, 4, 8]; they usually require a translation of the equivalence checking problem into a Boolean formula in conjunctive normal form (CNF) (union of clauses) [9]. In this paper we will focus on SAT based combinational equivalence checking.

Apart from methods which exploit structural similarities between the implementation and the specification [10–15], which will be discussed later on, there are basically two approaches for SAT based equivalence checking of two combinational circuits with m outputs (i.e. specification and implementation):

1. translating the implementation and the specification into *one* CNF which is satisfiable if and only if the implementation and the specification are not equivalent,
2. performing a translation into a corresponding CNF for each output separately leading to m different satisfiability checking problems.

Option 1) has the advantage that the SAT solver is able to exploit logic sharing between different output functions. But for large circuits it will produce large CNF representations leading to difficult problems for the SAT solver. For large circuits, where option 1) will lead to complexity problems, option 2) should be preferred. However, then a straightforward application of a SAT solver has the disadvantage that identical parts of the circuit are processed more than once, if logic sharing between the cones of different outputs exists.

For this reason, we will produce separate CNFs for each output, and we make use of an *incremental* SAT solver. Incremental SAT solvers [16, 17] are able to solve several SAT problems $S_1, \ldots S_n$ one after the other, if the CNF of S_{i+1} results from the CNF of S_i by adding clauses. The solution to a problem S_{i+1} in an incremental SAT solver may profit from the knowledge obtained during the solution to problems $S_1, \ldots S_i$ (this knowledge is represented by so-called conflict clauses). Currently, incremental SAT solvers are mainly used in the context of Bounded Model Checking (BMC) [18, 19] for property checking. Incremental SAT problems naturally arise during BMC: BMC applied to certain temporal properties (invariants or, more generally, LTL formulas) 'unfolds' the transition relation of a system for k steps in order to find a counterexample to the property. The BMC instance is translated into a SAT problem and, if no counterexample of length k is found by the SAT solver, k is increased and the BMC procedure is used again. Since CNF representations of unfoldings of the transition relation for $k' > k$ steps include the unfoldings for k steps, BMC can certainly profit from incremental SAT techniques [20, 21].

In general SAT based equivalence checking is used with separate CNFs for each output function, earlier SAT problems will not be completely contained in later SAT problems. However, there will be an overlap between different SAT problems due to logic sharing between different output functions. For this reason, we add the corresponding CNFs to the incremental SAT solver output by output and check for each output separately. During the SAT check of a later output the incremental SAT solver is able to profit from conflict clauses learnt in the SAT checks for earlier outputs. In this paper we present heuristics which try to maximize the benefit from incremental SAT solving in combinational equivalence checking by looking for

good orders in which the equivalence of outputs is checked. Our experiments prove that incremental SAT together with an optimized output ordering is able to significantly improve run times.

Moreover, we provide a reset strategy which removes all clauses (original clauses and conflict clauses) of previous output functions (i.e., which resets the incremental SAT solver), if we no longer expect to profit from the incremental SAT strategy. This is the case when the clause database becomes large and the number of shared clauses between the currently inserted output function and the existing clause data base is small. Using the reset strategy the experimental results have been improved further.

Our approach using incremental SAT with output ordering and resets may be used in cases when the specification and the implementation do not show any structural similarities. However, if structural similarities are present, methods which detect equivalences between internal nodes should be used in order to simplify the overall equivalence checking problem [10–15]. In applications like [7, 22], simulation is used for detecting pairs of candidate nodes for equivalence. Then 'SAT sweeping' detects whether these pairs of candidates are really equivalent or not. The generalization of our method to this scenario is straightforward: In this case, our method will be applied to the pairs of equivalence candidates rather than to pairs of outputs of specification and implementation.

The paper is structured as follows: In Section II we give a brief review of combinational equivalence checking, SAT and incremental SAT. Section III gives an overview of our approach, whereas sections IV and V describe our output ordering heuristics and the reset strategy in more detail. Section VI presents experimental results proving the efficiency of our approach, and finally, Section VII concludes the paper.

II. Preliminaries

A. Combinational Equivalence Checking of Circuits

A combinational circuit C with n primary inputs and m primary outputs represents a boolean function $f^C : \mathbb{B}^n \rightarrow \mathbb{B}^m$.

For the SAT based equivalence checking of two circuits, C_A and C_B, usually the corresponding primary outputs are connected by a miter structure which is simply an XOR-gate. The corresponding primary inputs of the two circuits are connected, i.e., the two circuits are depending on the same inputs. The resulting circuit has one miter output for every pair of corresponding outputs. See Fig. 1 for an illustration. To disprove equivalence it is necessary to find at least one miter output which evaluates to 1 for an arbitrary input assignment, i.e., it is necessary to find an input assignment where the two outputs evaluate to different values.

This can be achieved by checking one miter after another (single output approach, see Fig. 1) or by a disjunction over all miter outputs to check all miters in parallel (all outputs approach, see Fig. 2). Before applying a CNF based SAT solver for solving the problem the resulting circuits have to be transformed into conjunctive normal form (CNF). This is typically done by doing a Tseitin transformation [9], which introduces auxiliary variables for every signal in the circuits. The size of the resulting CNF is linear w.r.t. the size of the corresponding circuit.

For completeness we give the following definitions and an example:

Definition 1 *Let V be the set of boolean variables. Then the expression b^ϵ with $b \in V$ and $\epsilon \in \{0, 1\}$ is called a* literal, *where $b^0 \equiv \overline{b}$ and $b^1 \equiv b$ holds. A* clause *is a disjunction of literals (e.g. $c = (\overline{b_1} + b_2 + b_3)$ is a clause c with the Boolean variables b_1,b_2 and b_3). A* Conjunctive Normal Form (CNF) *is a conjunction of clauses.*

The Tseitin transformation of the half adder in Fig. 3 e.g. is performed in the following way: The AND gate leads to the CNF $(\overline{s_1} + a)(\overline{s_1} + b)(s_1 + \overline{a} + \overline{b})$ (= '$s_1 \equiv a \cdot b$') and the XOR gate is represented by $(\overline{s_2} + a + b)(\overline{s_2} + \overline{a} + \overline{b})(s_2 + \overline{a} + b)(s_2 + a + \overline{b})$ (= '$s_2 \equiv (a \oplus b)$') The CNF resulting from the conjunction of both descriptions evaluates only to 1, if a consistent assignment to all variables (input variables and auxiliary variables) is applied.

After transforming circuits with miter structures (see Figures 1 and 2) to CNF, we have to ensure that we only look for consistent assignments which produce the value 1 at the output of the miter (single output approach) or at the output of the disjunction of miters (all output approach). Assuming that the auxiliary variable for this output is s_j, this can be achieved by simply adding a unit clause s_j to the CNF. The resulting CNF is satisfiable iff the two circuits are not equivalent.

Checking the equivalence of all outputs at once or separately for single outputs has specific advantages and disadvantages:

A.1 Single Output Approach (SOA)

The whole CEC problem is split into pieces, that are solved one after another. Sometimes this is the only way to handle big problem instances, where the whole problem exceeds memory or time limits. Even when some of the sub-problems are too hard for the SAT solver, normally a partial verification of some outputs can be done. One disadvantage of this method is, that structural sharing between the different sub-problems is not exploited. Therefore the SAT solver has to process identical parts of the CNF repeatedly, duplicating Boolean reasoning already performed before. When the circuit contains a large amount of shared structures between different output cones, this may result in a higher overall runtime.

A.2 All Output Approach (AOA)

Here Boolean reasoning performed for shared subcircuits can be reused for several outputs. However due to the nature of SAT, handling big problem instances may be hard, thus the advantage of sharing structure potentially does not pay off, leading to high run times for SAT solving.

B. SAT

Most modern CNF based SAT solvers implement the DPLL Algorithm [2, 23] to solve the satisfiability problem for a given CNF. The basic algorithm performs in a competitive way with other methods in CEC only if it is supplemented by additional techniques which have been developed during the last few years. Most current state-of-the-art SAT solvers include nonchronological backtracking [3], conflict clause learning [3, 24], improved decision heuristics (e.g. VSIDS [4]), search restarts [4, 25] and the two-literal-watching scheme [4].

C. Incremental SAT

To support the special structure of SAT problems which arise for example in BMC several modern solvers implement also a technique called *incremental SAT solving* [26]. Conventional

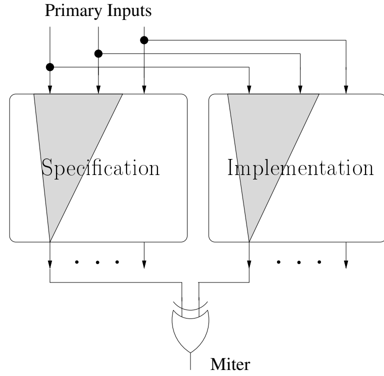

Fig. 1. Single Output Approach

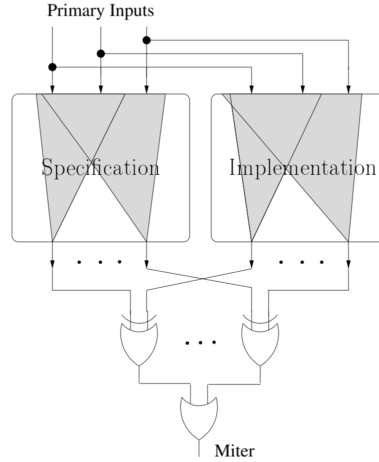

Fig. 2. All Output Approach

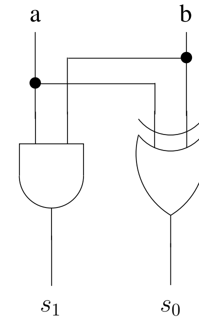

Fig. 3. Example: Half Adder

SAT solvers prove whether a problem encoded in CNF is satisfiable or not and, if a second SAT problem is solved afterwards, they restart from scratch. If the next problem instance shares parts of the CNF with the previous problem, information already obtained in the previous step is lost (e.g. learned conflict clauses, activity values). In contrast, incremental SAT solvers preserve their internal state after a SAT run, allowing additional CNF structures to be added after each run.

When incremental SAT is applied in applications like BMC, it is not only necessary to be able to add clauses after each step, but also to delete or disable clauses: After a step k which fails to disprove a property by a counterexample of length k, clauses for step $k+1$ have to be added, and clauses describing the property checked in step k have to be removed for the following steps (otherwise the following SAT problems would be unsatisfiable, too).

Two approaches are used to handle this problem: The first approach provides special measures to delete a set of clauses from the clause database, including the removal of all conflict clauses depending on the deleted clauses. (This approach is used in zChaff [4].) The second approach (used in the solver MiniSAT [8]) is based on additional activity variables which may be set by so-called assignments. For example a clause $c = (l_1 + ... + l_n)$ with literals $l_1, ..., l_n$ can be extended by an activity variable a leading to clause $(\bar{a} + l_1 + ... + l_n)$. If the activity variable a is assigned to 1, the literal $\bar{a}$ is false and the extended clause reduces to $(l_1 + ... + l_n)$ again. If a is assigned to 0, then the clause evaluates to true, thus the clause is deactivated. The assignments to activity variables are performed before the SAT solver is run, and they are always kept until the solver returns the result. In this way, clauses (and all conflict clauses dependent upon them) may be effectively removed before a SAT solver run [8].

III. Our Approach

We propose an extension of the Single Output Approach (SOA) using incremental SAT solving. In this way, we combine the advantage of SOA (smaller SAT instances) with the advantage of AOA (sharing of structures). To achieve this goal we developed heuristics to guide the overall process.

The first heuristic method defines the order of the equivalence checks. This is due to the fact that the order of the outputs in the circuit definitions may be arbitrarily bad or good for the incremental approach. The basic idea here is to start with simple problems and consecutively solve more and more difficult problems. The following problems should ideally share as much of the structure as possible to support the incremental SAT approach.

The second heuristic method is responsible for a complete reset of the SAT solver before a new output pair is checked. This may be necessary because the different SAT problems do not necessarily share a reasonable amount of nodes. For instance, a circuit with two completely independent structures will not profit from incremental SAT, because the solution to one sub-problem does not benefit from information learnt during the solution of the other. In this case it is better to start a completely new SAT instance, to prevent the processing of unnecessary clauses. (This is a difference to the SAT problems occurring in BMC, where the following SAT problems contain all previous SAT problems besides the property checking structure.)

In our approach we use a SAT solver which supports variable assignments in every run [8]. In our case we do not need additional activity variables for removing clauses corresponding to miters of previous equivalence checks: We do not add unit clauses forcing miter outputs to a constant 1 in our approach, but we assign variables before SAT solver runs instead. To activate the miter of the current output pair being checked, it is sufficient to assign the miter output variable to a constant 1 in this step. Then the SAT solver can only find a model where a boolean difference between the outputs exists. Since we do not perform assignments to miter outputs of previous checks, these miters do not hurt correctness. There always exists a consistent assignment to that miter signals, since the output can be freely assigned to 0 or 1 by the SAT solver. In this way the boolean reasoning for parts of the circuits only connected to the previous pairs of outputs is limited, since backtracks forcing the miter outputs of previous checks to constant 1 will not occur. (Either way, remember that our reset strategy will reset the SAT solver whenever the size of the clause representation in the SOA approach will be too large and we do not expect to profit much from the previously learnt conflict clauses.)

The main CEC routine (see Fig. 4) has two input operands, the two circuits which are checked for equivalence. It returns whether the two circuits are equivalent or not as a boolean value.

In the first step, the two circuits are merged and miters are added by the subroutine `add_miters(A,B)`. Thereafter the outputs of the miters are ordered according to the ordering heuristics. Then the main loop is entered. It checks the miters using the order which was created by the ordering heuristics.

At the beginning of the main loop the function `threshold_exceeded(C.output(i))` is called. It returns true when the current CEC problem does not share enough with previous problems which are still present in the solver. In this case a reset of the SAT solver is executed. Afterwards the SAT solver does not contain any clause or other information. After the reset the clauses corresponding to the current pair of outputs are added.

Thereafter the output of the checked miter is set to true by an assignment to the solver and the SAT procedure is started. If this is successful, i.e. the solver returns true, then a inequality of two outputs is found and the for-loop is exited. Otherwise the next miter is checked. Equality of the two circuits is found when all SAT checks were unsatisfiable.

In this section we assume that our approach is applied to the primary outputs of the implementation and the specification. However, note that a generalization of our approach to the case where there are structural similarities between implementation and specification is straightforward: In that case usually And-Inverter graph (AIG) representations for the implementation and the specification are constructed [15, 22]. During the construction of AIGs local transformation rules like one-level or two-level structural hashing are used in order to identify equivalences between internal nodes which are easy to detect. Then, a simulation of input vectors provides further pairs of nodes which are candidates for equivalence. A following 'SAT sweeping' run detects whether these pairs of candidate nodes are really equivalent or not. In this scenario our method will provide an order in which these pairs of candidate nodes are presented to the SAT solver and a reset strategy for resetting the SAT solver during SAT sweeping. In our algorithm 'pairs of outputs of the specification and the implementation' just has to be replaced by 'pairs of candidate nodes for equivalence'.

Fig. 4. Main CEC Routine

```
enum Result { EQUAL, UNEQUAL };

Result CEC(Circuit A, Circuit B)
{
    bool solver_result = false;
    SAT_Solver solver;

    // Preprocessing
    C = add_miters(A,B);

    // apply new order
    order_outputs(C);

    // Main loop
    for(int i=1; i <= C.nr_of_outputs(); ++i)
    {
        if( threshold_exceeded(C.output(i)) )
        {
            // Current sub-circuit shares not enough
            // structure with its predecessors
            Solver.reset();
        }

        // add current output cone
        solver.add( C.cone(i) );

        // assign miter output i to true
        solver.assign( C.output(i), true)

        // start solver and store the result
        solver_result = solver.solve();

        if( solver_result == true )
        {
            // Counterexample found
            break;
        }
    }

    // return the result of CEC
    if( solver_result == true )
    {
        return UNEQUAL;
    }
    return EQUAL;
}
```

IV. OUTPUT ORDERING HEURISTICS

As described above, the basic idea is to order the miter outputs in a way that assists the incremental SAT attempt. The new miter output sequence is defined as follows.

Definition 2 *Let C^f be the set of clauses which represents the cone of output f (as described in Section II.A [9]). The set $C_{i,j}$ denotes the set of clauses which represent the cones of the outputs occurring from position i up to j in the new order, with $1 \leq i \leq j \leq m$, where m is the number of miter outputs.*

The order is computed iteratively beginning with position 1. In the first step we choose the output f where $|C^f|$ is minimal, i.e. we start with the smallest cone. Then we select in every following step $j+1$ one of the remaining miter outputs where $|C^f \setminus C_{1,j}|$ is minimal.

Thus we will always start with small problems and minimize the count of new clauses which are added at each step to the SAT solver during the CEC routine. Due to the nature of SAT problems, it is better to solve small problems first and then reuse the learnt information in larger problems.

This heuristic approach also favors an order where outputs which have cones with a lot of sharing appear nearby in the order and therefore supports the incremental approach. [1]

V. RESET STRATEGY

In this section we describe a heuristic approach to decide whether or not to reset the SAT solver completely during a sequence of incremental SAT solver runs.

Definition 3 *Let f_j be the output f at position j of the ordering, with $1 \leq j \leq m$. Let the position of the last reset of the incremental SAT solver be immediately before output i ($1 \leq i < j$), i.e. the incremental SAT solver currently contains clauses for outputs $f_i, ..., f_{j-1}$. Initially i is set to 1. Then the* reset ratio *R_j of output f_j is defined as follows:*

$$R_j := \frac{|C^{f_j} \cap C_{i,j-1}|}{|C_{i,j-1}|}$$

The reset strategy considers all clauses which are present at the moment in the SAT solver ($C_{i,j-1}$) and the clauses of the current CEC problem which are already present in the SAT solver ($C^{f_j} \cap C_{i,j-1}$). If the ratio is below a certain threshold

[1] In the extreme case of all cones being completely disjoint this method will lead to a sorting of the outputs depending on the cone size. Due to the reset heuristics described in the next section our approach then reduces to SOA. The only overhead is due the computation time of the heuristics, which is negligible w.r.t. the complexity of SAT.

$0 \leq t \leq 1$ a reset of the SAT-solver is performed. The variable i is then updated to the value of j.

For instance when $t = 0.2$ and $R_j = 0.15$ holds then a reset is triggered because the current CEC problem includes only a fraction of 15 percent of the previous problems which are still present in the solver, but 20 percent are at least needed.

If the threshold t is 0 then no reset is performed at all. A threshold > 1 would perform a reset in every step, which is equivalent to the SOA. Thus this parameter t indicates how aggressive the SAT solver is reset.

VI. Experiments

A. Experimental Setting and Benchmarks

Our approach has been implemented using MiniSAT 1.14 [8]. All experiments were performed on a Linux desktop with a 2.8 GHz Intel Pentium IV processor with 1 GB of memory. Every single experiment had a timeout of 7000 seconds.

In our experiments we used two public available benchmark sets. The first is the ISCAS85 benchmark set [27], the second is the ITC99 benchmark set [28], where we used the BLIF version of the Torino subset. The second set contains only sequential circuits. To perform CEC we extracted the combinational part of them (we added to the circuit name an extension '`.c`' to indicate this).

In our experiments we confined ourselves to pairs of *equivalent* CEC examples which form the harder case for SAT solving. For practical applications we recommend using a simulation first in order to filter out erroneous designs where errors are easy to detect. If simulation does not detect an error and the benchmark size is not as large as to prevent the application of the AOA approach, a AOA run with a strict limit on run times should follow. This run may filter out erroneous designs with errors which are not too hard to detect, since the AOA approach looks for errors of different outputs in parallel, such that it may be fast, if there is an error for some 'easy output'. The approach presented here should be applied only after filtering out these easy cases.

To obtain for every circuit a different but equivalent counterpart we used SIS [29] with the `simplify` command. Versions processed by SIS are marked by '`.s`' at the end of the circuit name. Since we did not spend much effort of applying more powerful logic synthesis algorithms, we compared two versions of a circuit which are rather similar from their structure. However, note that we could have also made use of equivalent counterparts without any structural similarities, if they had been at our disposal, since our current method does not exploit structural similarities.

B. Experimental Results

The results of our experiments are listed in the Table VI. Run times are given in CPU seconds. In the first two columns the names of the circuits are shown. Column 3 and 4 are presenting the run times for AOA and SOA. Columns 5 up to 10 contain our incremental approach with the ordering and reset heuristics. We used six different values for the threshold t of the reset strategy. In Column 11 ('ordering, no reset') we used only ordering of the outputs. This corresponds to $t = 0$. The last column shows the results of a straightforward incremental approach, where no ordering and reset is used. The sequence of equivalence checks was defined here by the original output ordering of the BLIF file. For all benchmarks and approaches we computed the speedup relative to the AOA run times. In the last row we show the respective averages of speedups over all benchmarks. (For cases when a timeout occurred these values are only a lower bound for the real speedup.) The row before shows the sums of run times for all benchmarks.

In the Table I we omitted benchmarks of the benchmark set where for all methods run times of single experiments were below 0.3 CPU seconds.

C. Discussion

The comparison of AOA and SOA confirmed our expectations: AOA has advantages in smaller experiments, where the overall complexity of the SAT problem is not too big and the reuse of information pays off. When examples become larger, SOA does better.

The use of incremental techniques without reordering and reset (last column) leads to considerable speedups compared to the traditional approaches, the average speedup compared to AOA is 7.35.

These results could even be improved to a large extent by our output ordering heuristics (column 11, 'ordering, no reset'): Without output ordering the run times are 210% higher for C5315 and 112% higher for b21, e.g. . The average speedup compared to AOA improved from 7.35 to 13.52. The results demonstrate that output ordering, taking both the sizes of the output cones and sharing between output cones into account clearly pays off.

Although many benchmarks in the benchmark set contain considerable amounts of logic sharing, our reset strategy is able to provide further improvements in most cases. The overall run time of 2952.62 s for the version without reset could be improved to an overall run time of 2760.18 s for the version with reset and threshold $t = 0.4$. For the C6288 benchmark (a multiplier circuit) however, no reset is performed at all and all run times are in the same range. This is due to the high amount of sharing in the multiplier circuit. For other examples like benchmark b17.c which contains more disjoint structure, run times could be improved by resets (The number of resets ranges between 92 (for $t = 0.1$) and 278 (for $t = 0.6$) for b17.c, a circuit with 1512 outputs.). For the set of benchmarks used in this paper the selection of $t = 0.4$ seems to perform best, increasing the threshold value even more tends to be counterproductive (the behavior for $t \rightsquigarrow 1.0$ corresponds to the single output approach (SOA)). We expect that the reset strategy will become much more important when we consider larger examples which contain more parts which are more or less independent from each other.

VII. Conclusion and Future Work

We presented a novel approach to solve the CEC problem with SAT based techniques. The results clearly show that we are able to get significant speedups w.r.t. to traditional techniques when incremental SAT is supplemented by clever heuristics. In the future we expect to obtain an even more precise reset strategy by a tighter integration of our heuristics into a SAT solver which would it make possible to make use of more information, e.g., about the presence of conflict clauses in the clause database for certain parts of the circuit. Moreover, we will evaluate a generalization of our approach to the context of exploiting structural similarities by And-Inverter Graphs and SAT sweeping [15,22,30] (see also Section III). We expect that the incremental SAT approach will profit from a higher sharing of circuit structures in an And-Inverter Graph based intermediate representation.

TABLE I
EXPERIMENTAL RESULTS

Benchmark		CPU Time (s)									
				Incremental Approaches							
Circuit A	Circuit B	AOA	SOA	Reset and Ordering						Ordering	No Ordering
				$t = 0.6$	$t = 0.5$	$t = 0.4$	$t = 0.3$	$t = 0.2$	$t = 0.1$	No Reset	No Reset
C499	C499.s	0.53	2.34	0.23	0.15	0.15	0.15	0.15	0.14	0.16	0.16
C880	C880.s	0.38	0.24	0.11	0.12	0.09	0.11	0.09	0.09	0.08	0.09
C1355	C1355.s	0.37	3.82	0.30	0.29	0.31	0.31	0.29	0.30	0.32	0.27
C1908	C1908.s	0.51	4.19	0.45	0.43	0.43	0.44	0.42	0.42	0.46	0.40
C2670	C2670.s	0.80	1.44	1.11	1.04	1.06	0.91	0.92	0.87	0.38	1.07
C3540	C3540.s	11.78	38.99	4.18	2.86	2.35	3.56	1.36	1.14	1.28	1.80
C5315	C5315.s	6.69	16.60	3.63	3.70	3.68	3.33	3.49	3.03	1.92	5.96
C6288	C6288.s	>7000	>7000	2058.37	2048.43	2033.84	2045.77	2048.42	2062.81	2052.85	2859.98
C7552	C7552.s	17.86	4.96	3.03	2.98	3.20	3.10	2.94	2.93	4.61	4.32
b04.c	b04.c.s	0.40	0.71	0.33	0.31	0.30	0.30	0.38	0.48	0.41	0.40
b05.c	b05.c.s	0.91	1.68	0.49	0.48	0.48	0.43	0.49	0.46	0.47	0.56
b11.c	b11.c.s	0.42	0.52	0.36	0.33	0.33	0.35	0.32	0.29	0.30	0.32
b12.c	b12.c.s	0.79	0.83	0.55	0.49	0.43	0.43	0.45	0.43	0.53	0.81
b14.c	b14.c.s	3465.80	792.30	56.10	70.80	46.47	46.83	45.12	49.52	37.71	87.53
b15.c	b15.c.s	151.20	355.24	64.63	63.31	55.02	55.84	55.62	48.36	50.78	67.60
b17.c	b17.c.s	749.75	1156.20	214.25	179.19	166.38	171.72	166.08	172.35	253.17	351.24
b20.c	b20.c.s	>7000	2893.42	155.41	121.22	121.35	126.01	126.01	154.14	173.55	272.41
b21.c	b21.c.s	>7000	3186.40	177.62	144.61	142.43	142.15	135.50	161.30	137.31	291.30
b22.c	b22.c.s	>7000	4229.34	227.44	183.95	181.88	182.59	185.05	186.67	236.33	441.80
Overall runtime		>32408.19	>19689.22	2968.59	2824.69	2760.18	2784.37	2773.10	2845.73	2952.62	4388.02
Average Speedup:		1	1.18	11.12	12.16	13.69	13.44	14.01	12.80	13.52	7.35

REFERENCES

[1] R. Bryant, "Graph - based algorithms for Boolean function manipulation," *IEEE Trans. on Comp.*, vol. 35, no. 8, pp. 677–691, 1986.

[2] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," *Communications of the ACM*, vol. 5, pp. 394–397, 1962.

[3] J. Marques-Silva and K. Sakallah, "GRASP – a new search algorithm for satisfiability," in *Int'l Conf. on CAD*, 1996, pp. 220–227.

[4] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engeneering an efficient SAT solver," in *Design Automation Conf.*, 2001.

[5] P. Tafertshofer, A. Ganz, and M. Henftling, "A SAT-based implication engine for efficient ATPG, equivalence checking, and optimization of netlists," in *Int'l Conf. on CAD*, 1997, pp. 648 – 655.

[6] J. Marques-Silva and T. Glass, "Combinational equivalence checking using satisfiability and recursive learning," in *Design, Automation and Test in Europe*, 1999, pp. 145–149.

[7] E. Goldberg, M. Prasad, and R. Brayton, "Using SAT for combinational equivalence checking," in *Int'l Workshop on Logic Synth.*, 2000, pp. 185–191.

[8] N. Een and N. Sörensson, "An extensible sat-solver," in *Theory and Applications of Satisfiability Testing, 6th InternationalConference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8,2003 Selected Revised Papers*, ser. Lecture Notes in Computer Science, vol. 2919. Springer, 2003, pp. 541–638.

[9] G. Tseitin, "On the complexity of derivations in propositional calculus," in *Studies in Constructive Mathematics and Mathematical Logics*, A. Slisenko, Ed., 1968.

[10] J. Jain, R. Mukherjee, and M. Fujita, "Advanced verification techniques based on learning," in *Design Automation Conf.*, 1995, pp. 420–426.

[11] S. Reddy, W. Kunz, and D. Pradhan, "Novel verification framework combining structural and OBDD methods in a synthesis environment," in *Design Automation Conf.*, June 1995, pp. 414–419.

[12] Y. Matsunaga, "An efficient equivalence checker for combinational circuits," in *Design Automation Conf.*, 1996, pp. 629–634.

[13] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Design Automation Conf.*, 1997, pp. 263–268.

[14] J. Burch and V. Singhal, "Tight Integration of Combinational Verification Methods," in *Int'l Conf. on CAD*, 1998, pp. 570–576.

[15] V. Paruthi and A. Kuehlmann, "Equivalence checking combining a structural SAT-solver, BDDs, and simulation," in *Int'l Conf. on Comp. Design*, 2000, pp. 459–464.

[16] J. Whittemore, J. Kim, and K. Sakallah, "SATIRE: A New Incremental Satisfiability Engine," in *Design Automation Conference*, June 2001, pp. 542–545.

[17] H. Jin, M. Awedh, and F. Somenzi, "CirCUs: A satisfiability solver geared towards bounded model checking," in *Sixteenth Conference on Computer Aided Verification (CAV'04)*, ser. LNCS, vol. 3114. Springer-Verlag Heidelberg, July 2004, pp. 519 – 522.

[18] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 1579. Springer Verlag, 1999.

[19] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," in *Design Automation Conf.*, 1999.

[20] O. Strichman, "Tuning SAT checkers for Bounded Model Checking," in *Int'l Conf. on CAV*, 2000.

[21] N. Een and N. Sörensson, "Temporal induction by incremental sat solving," in *BMC'2003*, vol. 89:. Elsevier, 2003, pp. 541–638.

[22] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking," in *Int'l Conf. on Computer-Aided Design*, 2004, pp. 50–57.

[23] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *Journal of the ACM*, vol. 7, no. 3, pp. 201–215, 1960.

[24] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver," in *Int'l Conf. on CAD*, San Jose, CA, November 2001.

[25] L. Baptista, I. Lynce, and J. Marques-Silva, "Complete search restart strategies for satisfiability," IJCAI'01 Workshop on Stochastic SearchAlgorithms (IJCAI-SSA), August 2001.

[26] J. N. Hooker, "Solving the incremental satisfiability problem," *Journal of Logic Programming*, vol. 15, no. 1-2, pp. 177–186, 1993.

[27] F. Brglez and H. Fujiwara, "A neutral netlist of 10 combinational circuits and a target translator in fortran," in *Int'l Symp. Circ. and Systems, Special Sess. on ATPG and Fault Simulation*, 1985, pp. 663–698.

[28] F. Corno, M. S. Reorda, and G. Squillero, "RT-level ITC 99 benchmarks and first ATPG results," *IEEE Design & Test of Computers*, vol. July-August 2000, pp. 44–53, 2000.

[29] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," University of Berkeley, Tech. Rep., 1992.

[30] A. Kuehlmann, V. Paruthi, F. Krohm, and M. M.K. Ganai, "Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification," *IEEE Trans. on CAD*, 2002.