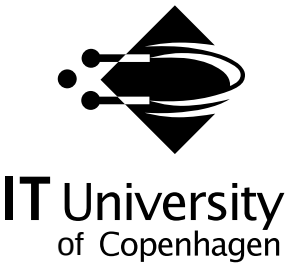

Satisfiability Checking Using Boolean Expression Diagrams

Poul Frederick Williams
Henrik Reif Andersen
Henrik Hulgaard

IT University Technical Report Series **TR-2000-1**

ISSN 1600–6100 **October 2000**

ISSN 1600–6100

ISBN 87–7949–001–8

Copies may be obtained by contacting:

IT University of Copenhagen
Glentevej 67
DK-2400 Copenhagen NV
Denmark

Telephone: +45 3816 8888
Telefax: +45 3816 8899
Web www.it-c.dk

Satisfiability Checking Using Boolean Expression Diagrams

Poul Frederick Williams[1] Henrik Reif Andersen[2] Henrik Hulgaard[3]

Abstract

In this paper we present a method for determining satisfiability of formulae represented by Boolean Expression Diagrams. The method uses the UP_ONE algorithm for splitting on variables and rewriting rules instead of unit propagation. We show how to combine the method with BDD construction. In this way our method can be seen as bridging the gap between standard SAT-solvers and BDD construction.

1 Introduction

In this paper we address the problem of determining satisfiability of formulae which are *not* on conjunctive normal form (CNF). One area where such formulae arise is formal verification. For example, in equivalence checking of combinational circuits we connect the outputs of the circuits with exclusive-or gates and construct a Boolean formulae for the combined circuits. The formulae is satisfiable if the two circuits are not functionally equivalent. Model checking without BDDs is another example where non-CNF formulae occur [1, 4, 5, 6, 21].

Boolean Expression Diagrams (BEDs) [2, 3] is a data structure for working with Boolean formulae. Given a BED for a formula, one way of proving satisfiability is to convert the BED to a Binary Decision Diagram (BDD) [9]. If the resulting BDD is the terminal **0** (a contradiction), then the formula is *not* satisfiable. Otherwise the formula is indeed satisfiable. Table 1 illustrates it for both satisfiability (SAT) and tautology (TAUT). There is the following relation between the two:

$$\text{SAT}\ \phi \quad = \quad \neg\text{TAUT}\ \neg\phi\,.$$

Formula	SAT	TAUT
Tautology	yes	yes
Contradiction	no	no
Neither	yes	no

Table 1: The table shows the result of solving the satisfiability (SAT) and tautology (TAUT) problem for three different types of formulae: tautologies (always 1), contradictions (always 0), and formulae which can take both values. For readability we use "yes" and "no" instead of 1 and 0.

Converting a BED into a BDD allows us to determine satisfiability. However, we obtain more information than just a "yes, the formula is satisfiable" or a "no, the formula is not satisfiable" answer. The resulting BDD encodes *all* possible variable assignments satisfying the formula. This is overkill as we are only interested in *one* of them – and sometimes just the *existence* of one.

It is possible to determine satisfiability for a formula represented by a BED by converting the BED to a CNF formula and then feed it to a SAT-solver like SATO [22] or GRASP [15]. Such SAT-solvers are very efficient in proving satisfiability. Unfortunately, the conversion from BED to CNF introduces k new variables, where k is the number of non-terminal vertices in the BED. It is possible to avoid the extra k variables, but at the expense of a potential exponential blowup in the formula size.

[1]Email pfw@it-c.dk
[2]Email hra@it-c.dk
[3]Email henrik@it-c.dk

2 Related Work

The satisfiability problem has been studied for a long time. The Davis-Putnam [11, 12] SAT-procedure has been known for around 40 years. It is still one of the best procedures for determining satisfiability.

People have also studied incomplete algorithms like Greedy SAT (GSAT) [17]. They are typically fast – often faster than the complete methods. However, they are incomplete which means that they are not always able to come up with an answer.

As mentioned in the introduction, most SAT-solvers expect the input formula to be on CNF. Giunchiglia and Sebastiani [13, 16] have examined GSAT and Davis-Putnam for use on non-CNF formulae. Stålmarck's method [18] also works on a non-CNF representation. Using non-CNF formulae alleviates the convertion into CNF and the addition of extra variables. However, some of these methods implicitly add the same number of extra variables anyway.

BDDs [9] and variations thereof [10] have until recently been the dominating data structures in the area of formal verification. Now people have begun studying the use of SAT-solvers. Biere *et al.* [4, 5, 6] introduce bounded model checking where SAT-solvers are used to find counterexamples of a given depth in Kripke structures. Abdulla *et al.* [1] and Williams *et al.* [21] study SAT-solvers in fixed-point iterations for model checking. Bjesse and Claessen [7] apply SAT-solvers to van Eijk's BDD-based method [19] for verification without state space traversal.

3 Boolean Expression Diagrams

A Boolean Expression Diagram [2, 3] is a data structure for representing and manipulating Boolean formulas. In this section we briefly review the data structure.

Definition 1 (Boolean Expression Diagram) *A* Boolean Expression Diagram *(BED) is a directed acyclic graph $G = (V, E)$ with vertex set V and edge set E. The vertex set V contains three types of vertices: terminal, variable, and operator vertices.*

- *A* terminal vertex v *has as attribute a value* $val(v) \in \{0, 1\}$.
- *A* variable vertex v *has as attributes a Boolean variable* $var(v)$, *and two children* $low(v), high(v) \in V$.
- *An* operator vertex v *has as attributes a binary Boolean operator* $op(v)$, *and two children* $low(v)$, $high(v) \in V$.

The edge set E is defined by

$$E \;=\; \left\{ (v, low(v)), (v, high(v)) \;\middle|\; v \in V \text{ and } v \text{ is a non-terminal vertex} \right\}.$$

The relation between a BED and the Boolean function it represents is straightforward. Terminal vertices correspond to the constant functions 0 and 1. Variable vertices have the same semantics as vertices of BDDs and correspond to the *if-then-else* operator $x \rightarrow f_1, f_0$ defined as $(x \wedge f_1) \vee (\neg x \wedge f_0)$. Operator vertices correspond to their respective Boolean connectives. This leads to the following correspondence between BEDs and Boolean functions:

Definition 2 *A vertex v in a BED denotes a Boolean function f^v defined recursively as:*

- *If v is a terminal vertex, then $f^v = val(v)$.*
- *If v is a variable vertex, then $f^v = var(v) \rightarrow f^{high(v)}, f^{low(v)}$.*
- *If v is an operator vertex, then $f^v = f^{low(v)}\ op(v)\ f^{high(v)}$.*

The BED data structure is a representation form for propositional logic. If we disallow operator vertices, we get a BDD.

There exist algorithms for transforming a BED into a BDD. One such algorithm is UP_ONE. It sifts variables one at a time to the root of the BED. Using UP_ONE repeatedly to sift all the variables transforms the BED to a BDD. Another algorithm is UP_ALL which constructs the BDD in a buttom-up way similar to the APPLY algorithm by Bryant [9]. We refer the reader to [2, 3, 14, 20] for a more detailed description of UP_ONE, UP_ALL and their applications.

4 Satisfiability using Conjunctive Normal Form Formulae

A CNF formula consists of a set of clauses. Each clause contains a number of literals, where a literal is either a variable or the negation of a variable. The literals within a clause are OR'ed together, whereas the clauses are AND'ed together.

The Davis-Putnam SAT-procedure [11, 12] works as described in Algorithm 4. Line 1 is the base case. Line 3 is the backtracking. Line 5 handles unit clauses, and line 8 and 9 handle splitting on literals. There are different heuristics for choosing a literal in line 8. One heuristic is to choose the literal in such a way that the assignments in line 9 produce the most unit clauses.

Name: DP ϕ
1: **if** ϕ is the empty set of clauses **then**
2: **return** 1
3: **else if** ϕ contains the empty clause **then**
4: **return** 0
5: **else if** a unit clause l occurs in ϕ **then**
6: **return** DP($assign(l, \phi)$)
7: **else**
8: $l \leftarrow$ *choose-literal*(ϕ)
9: **return** DP($assign(l, \phi)$) $\vee$ DP($assign(\neg l, \phi)$)

Algorithm 1: The basic version of Davis-Putnam. The function $assign(l, \phi)$ applies the truth value of literal l to the CNF formula ϕ. The function *choose-literal*(ϕ) picks a literal for DP to split on.

5 Satisfiability using Boolean Expression Diagrams

The basis of Davis-Putnam is a splitting on literals. In BEDs we can obtain the same effect by pulling a variable to the root using Up_One. After pulling a variable x up using Up_One, there are two situations:

- The new root vertex is a variable x vertex. Both *low* and *high* children are BEDs. The formula is satisfiable if either the *low* child or the *high* child (or both) represent a satisfiable formula.
- The new BED does not contain the variable x anywhere. The formula does not depend on x and we can pick a new variable to pull up.

If at any point we reach the terminal **1**, then we know that the formula is satisfiable. This suggests a recursive algorithm which pulls up variables one at a time. The test for the empty set of clauses (line 1 in Algorithm 4) becomes a test for the terminal **1**. The test for whether ϕ contains the empty clause (line 3) becomes a test for the terminal **0**. We cannot find unit clauses with BEDs. The unit clauses are used to reduce the CNF formula. Instead we use another type of reductions: BED rewriting rules [20]. Algorithm 5 shows the pseudo-code for the SAT-procedure BedSat.

The function *choose-variable* in line 6 of Algorithm 5 picks a variable to split on. With a clause form representation of the formula, it is natural to pick the variable in such a way as to obtain the most unit clauses after the split. This gives the most reductions due to unit propagation. We do not have a clause form formulae. However, we can still choose the variable as to get the most reductions. We perform the splitting using Up_One. In [20] we discussed different heuristics for picking good variable orderings for Up_One. The first variable in such an ordering would probably be a good variable to split on. In BedSat we do not need to split on the variables in the same order along different branches. We have chosen a simple implementation which does a depth-first traversal of the BED and picks the first variable it encounters.

In line 9 the algorithm forks in two: one fork for the low child and one fork for the high child. If a satisfying assignment is found in one fork, then it is not necessary to consider the other one. We have implemented a simple strategy of first examining the fork with the smaller BED size (least number of vertices). We do not have any a priori knowledge of which fork to choose so picking the smaller one makes sense as the runtime of Up_One depends on the the size of the BED.

```
Name: BedSat u
 1: if u = 1 then
 2:   return 1
 3: else if u = 0 then
 4:   return 0
 5: else
 6:   x ← choose-variable(u)
 7:   u' ← Up_One(x, u)
 8:   if u' is a variable x vertex then
 9:     return BedSat low(u')  ∨  BedSat high(u')
10:   else
11:     return BedSat u'
```

Algorithm 2: The BedSat algorithm. The argument u is a BED. The function *choose-variable*(u) picks a variable to split on.

Figure 1 shows graphically how BedSat works. The circles correspond to splitting points and the triangles correspond to parts of the BED which have (gray triangles) or have not (white triangles) been examined. The numbers next to the triangles indicate the size of the state space represented by each triangle assuming that there are n variables in total. At any point during the algorithm we can compute the fraction of the state space examined so far by adding the numbers from the gray triangles and dividing by the size of the complete state space 2^n.

For a user, seeing the percentage of the state space examined so far is nice. It is often frustrating to do a computation and not knowing whether it is just about to finish or it is going nowhere. With BedSat it is very easy to compute the percentage. Of course, the percentage does not say anything about the time remaining in the computation. However, it does allow us to detect wether we are making progress. One could even imagine a more sophisticated SAT-solver which jumps back a number of splits if the user felt that the current choice of split variables did not produce any progress. Or we could do it automatically by tracking how the percentage changes over time. No or little growth could indicate that we were picking the wrong sequence of variables to split on and we should consider backtracking and picking new split variables. We call such backtracking *premature* since we give up the current series of splits, back up, and try a new one. Premature backtracking is also done in many implementations of Davis-Putnam. It is geared toward satisfiable functions since we give up our search in a particular part of the state space and concentrate on other, and hopefully easier, parts. If we find a satisfying assignment in the easy part of the state space then we never need to revisit the difficult parts. For unsatisfiable functions we need to examine the whole state space, and giving up on one part now just postpones the problems. The only hope is that by choosing a different sequence of variables to split on, the rewriting rules collapse the difficult part of the state space.

As we perform more and more splits, we effectively assign more and more variables a value. The remaining BED shrinks. We could continue the splitting of variables until we were left with a terminal vertex. However, at some point it becomes easier just to convert the BED into a BDD from which we can decide whether the original formula is satisfiable (the BDD is *not* **0**) or we have to backtrack and continue spliting (the BDD is **0**).

We implement a *cutoff* size for the remaining BED; see Algorithm 5. If its size is less than the cutoff size then we convert it to a BDD. Otherwise we continue splitting. As we mentioned in the introduction, constructing the BDD is in some sense overkill as we calculate too much information. If we set the cutoff size too high we risk spending time doing all these extra computations. On the other hand, if we set the cutoff size too low we spend time doing a lot of splitting deep down in the BED where it would be faster to simply construct the BDD. In our examples, cutoff sizes between 100 and 400 vertices give good results. For the BDD construction we use Up_All with the Fanin variable ordering heuristic [20].

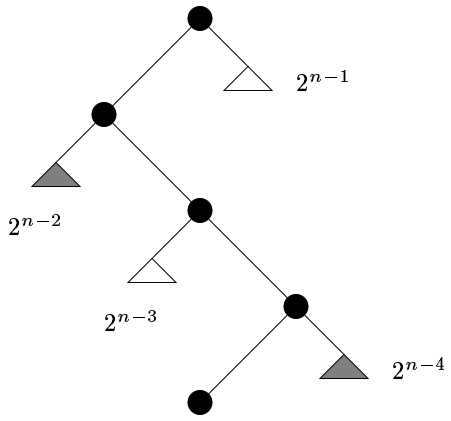

Figure 1: An illustration of the BEDSAT algorithm. Each circle represents a split on a variable. The top circle is the starting point. The triangles represent sub-BEDs; the white ones are as yet unexamined while the gray ones have already been examined. Assume that there are n variables in total and that the current position in BEDSAT corresponds to the bottom circle. Then the fraction of the state space which has already been examined is $\frac{2^{n-2}+2^{n-4}}{2^n}$.

Name: BEDSAT u
1: **if** $u = \mathbf{1}$ **then**
2: **return** 1
3: **else if** $u = \mathbf{0}$ **then**
4: **return** 0
5: **else if** $|u| <$ *cutoff-size* **then**
6: **return** (UP_ALL$(u) \neq \mathbf{0}$)
7: **else**
8: $x \leftarrow$ *choose-variable*(u)
9: $u' \leftarrow$ UP_ONE(x, u)
10: **if** u' is a variable x vertex **then**
11: **return** BEDSAT *low*(u') $\vee$ BEDSAT *high*(u')
12: **else**
13: **return** BEDSAT u'

Algorithm 3: The BEDSAT algorithm with cutoff *cutoff-size* measured in vertices. $|u|$ is the number of vertices in the BED u. Line 6 returns 0 if the BED u represents an unsatisfiable function and 1 otherwise.

6 Experimental Results

To see how well BEDSAT works in practice, we compare it to other techniques for solving SAT problems. The problems we use in the comparison are from the ISCAS'85 benchmark series [8] and from model checking [21]. All the problems have been turned into satisfiability problems.

All the experiments are performed on a 450 MHz Pentium III PC running Linux. All methods were limited to 32 MB of memory of 15 minutes of CPU time.

We compare BEDSAT to UP_ONE and UP_ALL with the FANIN variabel ordering heuristic. Furthermore, we compare with state-of-the-art SAT-solvers SATO and GRASP. Since both SATO and GRASP require their input to be in CNF form, we need to convert the BEDs to CNF. This increases the number of variables and thus also the state space for SATO and GRASP.

Table 2 shows the ISCAS'85 results. The first ten rows represent 475 unsatisfiable functions. We mark this with "U" in the second column. UP_ONE and UP_ALL (both with the FANIN ordering heuristic) work quite well. The SAT-solvers SATO and GRASP perform well on the smaller circuits (the number in the circuit names indicate the size), but give up on some of the larger ones. With 0 as cutoff size, BEDSAT does not perform well at all. The runtimes are an order of magnitude larger than the runtimes for the other methods. The long runtimes are due to BEDSAT's poor performance on some (but not all) unsatisfiable formulae. Increasing the cutoff size to 100 or 400 increases BEDSAT's performance. In fact, with cutoff size 400, BEDSAT yields runtimes comparable to or better than all other methods except the case of `c6288/nr` with UP_ONE.

The last five rows of Table 2 show the results for the erroneous circuits. Here there are 340 functions in total out of which 267 are unsatisfiable and 73 are satisfiable. We indicate this with "S/U" in the second column. The UP_ONE and UP_ALL methods take slightly longer on the erroneous circuits since not all BDDs collapse to a terminal. The SAT-solvers (SATO, GRASP and BEDSAT) perform strictly better on the erroneous circuits compared to the correct circuits; sometimes going from impossible to possible as for SATO and BEDSAT (with a cutoff size of 0 and 100) on `c7552`. BEDSAT is the only SAT-solver to handle `c3540` and it outperforms SATO and GRASP on `c5315`. On `c7552` BEDSAT is two orders of magnitude slower with a cutoff of 0, but yields comparable results when the cutoff size increases.

Consider the case of BEDSAT on `c3540` with a cutoff of 0. In the correct version, BEDSAT uses 185 seconds. This number reduces to 35.9 seconds for the erroneous version. The `c3540` example has 22 outputs where five are faulty in the erroneous version. BEDSAT has no problem detecting the errors in the five faulty outputs. In the correct version, about 149 seconds were spent on proving those five outputs to be unsatisfiable. Another example is `c1908` where, in the correct case, BEDSAT spends all its time (242 seconds) on one unsatisfiable output. In the erroneous version the difficult output has an error and the corresponding function becomes satisfiable. BEDSAT finds a satisfying assignment instantaneously (0.1 seconds). This indicates that BEDSAT is not very good at handling unsatisfiable formulae.

By varying the cutoff size, we can decide whether BEDSAT works mostly like regular BDD construction (a high cutoff size) or as a Davis-Putnam SAT-solver (a low cutoff size). It is worth noting that for `c3540` (both the correct and erroneous versions), BEDSAT with a cutoff of 400 gives better results than both pure BDD construction (UP_ALL) and standard SAT-solvers (SATO, GRASP, BEDSAT with 0 cutoff) by themselves.

Table 3 shows the results for the model checking problems. We have extracted satisfiability problems for the model checking experiments in [20]. The numbers 10, 20 and 30 indicate the output bit we are considering. The word "final" indicates the satisfiability problem for the final check of the specification. The word "last_fp" indicates the satisfiability problem for the last iteration in the fixed-point computation (where we detect that we have reached the fixed-point). The word "second_last_fp" indicates the satisfiability problem for the previous iteration in the fixed-point computation. The result column indicates whether the satisfiability problem is satisfiable (S) or not (U).

For the model checking problems, UP_ONE and UP_ALL perform very poorly. UP_ONE is only able to handle four out of 18 problems and UP_ALL only handles a single one. However, both UP_ONE and UP_ALL handle the `mult_10_final` problem which is difficult for the SAT-solvers. The SAT-solvers perform quite well – both on the satisfiable and the unsatisfiable problems. Most of the problems are solved in less than a second by all three SAT-solvers. While both SATO and GRASP take a long time on a few of the problems, BEDSAT seems to be more consistent in its performance.

Description	Result	UP_ONE	UP_ALL	SATO	GRASP	BEDSAT		
						0	100	400
c432/nr	U	2.1	1.7	0.5	0.4	36.4	3.5	1.4
c499/nr	U	4.3	1.8	1.8	1.4	17.8	16.7	1.7
c1355/nr	U	4.3	1.8	1.8	1.5	18.1	16.5	1.7
c1908/nr	U	0.7	0.6	0.4	0.4	242	11.1	0.2
c2670/nr	U	1.2	0.6	1.0	0.9	38.6	1.9	0.3
c3540/nr	U	32.3	39.2	-	-	185	133	10.9
c5315/nr	U	16.2	1.9	-	15.0	1.1	1.0	0.9
c6288/nr	U	2.7	-	-	-	-	-	-
c7552/nr	U	3.6	1.1	-	4.4	-	-	0.7
c1908/nr-err	S/U	0.7	0.6	0.4	0.4	0.1	0.1	0.2
c2670/nr-err	S/U	2.9	0.7	0.9	0.8	0.4	0.3	0.3
c3540/nr-err	S/U	42.8	40.2	-	-	35.9	15.8	4.6
c5315/nr-err	S/U	32.7	2.4	31.7	10.3	0.7	1.6	1.3
c7552/nr-err	S/U	8.1	1.8	2.5	2.6	176	2.0	1.3

Table 2: Runtimes in seconds for determining satisfiability of problems arising in verification of the ISCAS'85 benchmarks using different approaches. In the "Result" column, "U" indicates unsatisfiable problems while "S/U" indicates both satisfiable and unsatisfiable problems. Both UP_ONE and UP_ALL use the FANIN variable ordering heuristic. The last three columns show the results for BEDSAT. The numbers 0, 100 and 400 indicate the cutoff sizes in number of vertices. A dash indicates that the computation could not be done within the resource limits.

Description	Result	UP_ONE	UP_ALL	SATO	GRASP	BEDSAT
mult_10_final	U	13.1	43.5	-	-	31.9†
mult_10_last_fp	U	10.3	-	0.1	0.1	0.2
mult_10_second_last_fp	S	-	-	0.1	0.1	0.1
mult_20_final	?	-	-	-	-	-
mult_20_last_fp	U	-	-	0.1	0.1	0.1
mult_20_second_last_fp	S	-	-	0.5	40.9	0.5
mult_30_final	S	-	-	0.3	0.6	0.2
mult_30_last_fp	U	-	-	0.1	0.2	0.2
mult_30_second_last_fp	S	-	-	0.6	1.4	0.5
mult_bug_10_final	S	13.0	-	6.7	0.1	0.1
mult_bug_10_last_fp	U	9.9	-	0.1	0.1	0.2
mult_bug_10_second_last_fp	S	-	-	0.1	0.1	0.1
mult_bug_20_final	S	-	-	113	-	0.3
mult_bug_20_last_fp	U	-	-	0.1	0.1	0.1
mult_bug_20_second_last_fp	S	-	-	0.5	499	0.5
mult_bug_30_final	S	-	-	0.3	0.6	0.2
mult_bug_30_last_fp	U	-	-	0.1	0.2	0.2
mult_bug_30_second_last_fp	S	-	-	0.6	1.5	0.5

Table 3: Runtimes in seconds for determining satisfiability of problems arising in model checking using different approaches. In the "Result" column, "U" indicates unsatisfiable problems while "S" indicates satisfiable problems. Both UP_ONE and UP_ALL use the FANIN variable ordering heuristic. The BEDSAT experiments have cutoff size 0 (i.e., no cutoff) except for the one marked with † which has cutoff size 400. A dash indicates that the computation could not be done within the resource limits.

7 Conclusion

In this paper we have presented BEDSAT as an algorithm for solving the satisfiability problem on BEDs. Many traditional SAT-solvers require a CNF formula, but BEDSAT works directly on the BED and is thus able to take advantage of the data structure – for example by using the rewriting rules from [20] during the algorithm. Furthermore, BEDSAT avoids the conversion from BED into CNF which either adds extra variables or the CNF risks blowing up in size.

We have described how BEDSAT can work together with BDD construction yielding a better performance than both pure SAT-solvers and BDD construction by themselves. The combination works especially well on formulae which are unsatisfiable and thus difficult for pure SAT-solvers.

References

[1] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT solvers. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2000.

[2] H. R. Andersen and H. Hulgaard. Boolean expression diagrams. *Information and Computation.* (To appear).

[3] H. R. Andersen and H. Hulgaard. Boolean expression diagrams. In *IEEE Symposium on Logic in Computer Science (LICS)*, July 1997.

[4] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1999.

[5] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[6] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In *Computer Aided Verification (CAV)*, volume 1633 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[7] P. Bjesse. SAT-based verification without state space traversal. In *Proc. Formal Methods in Computer-Aided Design, Third International Conference, FMCAD'00, Austin, Texas, USA*, Lecture Notes in Computer Science, November 2000.

[8] F. Brglez and H. Fujiware. A neutral netlist of 10 combinational benchmarks circuits and a target translator in Fortran. In *Special Session International Symposium on Circuits and Systems (ISCAS)*, 1985.

[9] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.

[10] R. E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 236–243, November 1995.

[11] M. Davis, G. Longemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[12] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[13] E. Giunchiglia and R. Sebastiani. Applying the Davis-Putnam procedure to non-clausal formulas. In *Proc. Italian National Conference on Artificial Intelligence*, volume 1792 of *Lecture Notes in Computer Science*, Bologna, Italy, September 1999. Springer-Verlag.

[14] H. Hulgaard, P. F. Williams, and H. R. Andersen. Equivalence checking of combinational circuits using boolean expression diagrams. *IEEE Transactions on Computer Aided Design*, July 1999.

[15] J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48, 1999.

[16] R. Sebastiani. Applying GSAT to non-clausal formulas. *Journal of Artificial Intelligence Research (JAIR)*, 1:309–314, January 1994.

[17] B. Selman, H. J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In P. Rosenbloom and P. Szolovits, editors, *Proc. Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. American Association for Artificial Intelligence, AAAI Press.

[18] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In G. Gopalakrishnan and P. J. Windley, editors, *Proc. Formal Methods in Computer-Aided Design, Second International Conference, FMCAD'98, Palo Alto/CA, USA*, volume 1522 of *Lecture Notes in Computer Science*, pages 82–99, November 1998.

[19] C.A.J. van Eijk. Sequential equivalence checking without state space traversal. In *Proc. International Conf. on Design Automation and Test of Electronic-based Systems (DATE)*, 1998.

[20] P. F. Williams. *Formal Verification Based on Boolean Expression Diagrams.* PhD thesis, Dept. of Information Technology, Technical University of Denmark, Lyngby, Denmark, August 2000. ISBN 87-89112-59-8.

[21] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 124–138, Chicago, U.S.A., July 2000. Springer-Verlag.

[22] H. Zhang. SATO: An efficient propositional prover. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 272–275, Berlin, July 1997. Springer-Verlag.