Satisfiability Checking Using Boolean Expression Diagrams

Poul Frederick Williams, Henrik Reif Andersen, and Henrik Hulgaard

IT University of Copenhagen
Glentevej 67, DK-2400 Copenhagen NV, Denmark
{pfw,hra,henrik}@it-c.dk

Abstract. In this paper we present an algorithm for determining satisfiability of general Boolean formulas which are *not* necessarily on conjunctive normal form. The algorithm extends the well-known Davis-Putnam algorithm to work on Boolean formulas represented using Boolean Expression Diagrams (BEDs). The BED data structure allows the algorithm to take advantage of the built-in reduction rules and the sharing of sub-formulas. Furthermore, it is possible to combine the algorithm with traditional BDD construction (using Bryant's Apply-procedure). By adjusting a single parameter to the BedSat algorithm it is possible to control to what extent the algorithm behaves like the Apply-algorithm or like a SAT-solver. Thus the algorithm can be seen as bridging the gap between standard SAT-solvers and BDDs. We present promising experimental results for 566 non-clausal formulas obtained from the multi-level combinational circuits in the ISCAS'85 benchmark suite and from performing model checking of a shift-and-add multiplier.

1 Introduction

In this paper we address the problem of determining satisfiability of non-clausal Boolean formulas, i.e., formulas which are *not* necessarily on conjunctive normal form. One area where such formulas arise is in 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 formulas for the combined circuits. The formulas is satisfiable if the two circuits are not functionally equivalent.

Another important area in which non-clausal formulas arise is in model checking [1,4,5,6,24]. In bounded model checking, the reachable state space is approximated by (syntactically) unfolding the transition relation and obtaining a propositional formula which is not in clausal form. In order to check whether the approximated state space R violates a given invariant I, one has to determine whether the formula $\neg I \wedge R$ is satisfiable.

Boolean Expression Diagrams (BEDs) [2,3] is an extension of Binary Decision Diagram (BDD) [9] which allows Boolean operator vertices in the DAG. BEDs

T. Margaria and W. Yi (Eds.): TACAS 2001, LNCS 2031, pp. 39–51, 2001.

can represent any Boolean formulas in linear space at the price of being non-canonical. However, since converting a Boolean formula into a BDD via a BED can always be done at least as efficiently as constructing the BDD directly, many of the desirable properties of BDDs are maintained.

Given a BED for a formula, one way of proving satisfiability is to convert the BED to a BDD. The formula is satisfiable if and only if the resulting BDD is different from the terminal **0** (a contradiction). BDDs have become highly popular since they often are able to represent large formulas compactly. However, by converting the BED into a BDD, more information is obtained 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. In some cases this extra information is not needed since we may only be interested in *some* satisfying assignment or simply in whether such an assignment exists. The canonicity of BDDs also means that some formulas (such as the formulas for the multiplication function) cannot be efficiently represented and thus the approach to convert the BED to a BDD will be inefficient.

Instead of converting the BED to a BDD, one can use a dedicated satisfiability solver such as SATO [25] or GRASP [17]. These tools are highly efficient in finding a satisfiable assignment if one exists. On the other hand, they are often much slower than the BDD construction when the formula is unsatisfiable. Another problem with these algorithms is that the Boolean formula must be given in conjunctive normal form (CNF), and converting a general formula (whether represented as a BED or as a Boolean circuit) to CNF is inefficient: either k new variables are introduced (where k is the number of non-terminal vertices in the BED) or the size of the CNF may grow exponentially in the size of the formula.

The BEDSAT algorithm presented in this paper attempts to exploit the advantages of the two above approaches. The algorithm extends the Davis-Putnam algorithm to work directly on the BED data structure (thus avoiding the conversion to CNF). By using the BED representation, the algorithm can take advantage of the built-in reduction rules and the sharing of isomorphic sub-formulas. For small sub-BEDs (i.e., for small sub-formulas), it turns out that it is faster than running Davis-Putnam to simply construct the BDD and checking whether the result is different from **0**. In the BEDSAT algorithm, this observation is used by having the user provide an input N to the algorithm. When a sub-formula contains less than N BED vertices, the algorithm simply builds the BDD for the sub-formula and checks whether the result is different from the terminal **0**. When using $N = 0$, the BEDSAT algorithm reduces to an implementation of Davis-Putnam on the BED data structure (which is interesting in itself) and when using $N = \infty$, the BEDSAT algorithm reduces to Bryant's APPLY-algorithm for constructing BDDs bottom-up. Experiments show that the BEDSAT algorithm is significantly faster than both pure BDD construction and the dedicated satisfiability-solvers GRASP and SATO, both on satisfiable and unsatisfiable formulas, when choosing a value of N of 400 vertices.

Related Work

Determining whether a Boolean formula is satisfiable is one of the classical NP-complete problems and algorithms for determining satisfiability have been studied for a long time. The Davis-Putnam [11,12] SAT-procedure has been known for about 40 years and it is still considered one of the best procedures for determining satisfiability. More recently, incomplete algorithms like Greedy SAT (GSAT) [20] have appeared. These algorithms are faster than the complete methods, but by their very nature, they are not always able to complete with a definitive answer.

Most SAT-solvers expect the input formula to be in CNF. However, Giunchiglia and Sebastiani [13,19] have examined GSAT and Davis-Putnam for use on non-CNF formulas. Although these algorithm avoid the explicit conversion of the formula to CNF, they often implicitly add the same number of extra variable which would have been needed if one converted the formula to CNF. Stålmarck's method [21] is another algorithm which does not need the conversion to CNF.

BDDs [9] and variations thereof [10] have until recently been the dominating data structures in the area of formal verification. However, recently researchers have started studying the use of SAT-solvers as an alternative. Biere *et al.* [4,5,6] introduce bounded model checking where SAT-solvers are used to find counterexamples of a given depth in the Kripke structures. Abdulla *et al.* [1] and Williams *et al.* [24] study SAT-solvers in fixed-point iterations for model checking. Bjesse and Claessen [7] apply SAT-solvers to van Eijk's BDD-based method [22] for verification without state space traversal.

2 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 (false) and 1

(true). Variable vertices have the same semantics as vertices of BDDs and correspond to the *if-then-else* operator $x \to f_1, f_0$ 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 $f^v = var(v) \to f^{high(v)}, f^{low(v)}$.*
- *If v is an operator vertex, then $f^v = f^{low(v)}\ op(v)\ f^{high(v)}$.*

A BDD is simply a BED without operators, thus a strategy for converting BEDs into BDDs is to gradually eliminate the operators, keeping all the intermediate BEDs functionally equivalent. There are two very different ways of eliminating operators, called Up_All and Up_One. The Up_All algorithm constructs the BDD in a bottom-up way similar to the Apply algorithm by Bryant [9].

The Up_One algorithm is unique to BEDs and is based on repeated use of the following identity (called the *up-step*):

$$(x \to f_1, f_0)\ op\ (x \to f_1', f_0') = x \to (f_1\ op\ f_1'), (f_0\ op\ f_0')\,, \tag{1}$$

where op is an arbitrary binary Boolean operator, x is a Boolean variable, and f_i and f_i' $(i = 0, 1)$ are arbitrary Boolean expressions. This identity is used to move the variable x above the operator op. In this way, it moves operators closer to the terminal vertices and if some of the expressions f_i are terminal vertices, the operators are evaluated and the BED simplified. By repeatedly moving variable vertices above operator vertices, all operator vertices are eliminated and the BED is turned into a BDD. (Equation (1) also holds if the operator vertex op is a variable vertex. In that case, the up-step is identical to the level exchange operation typically used in BDDs to dynamically change the variable ordering [18].)

The Up_One algorithm gradually converts a BED into a BDD by pulling up variables one by one. The main advantage of this algorithm is that it can exploit structural information in the expression. We refer the reader to [2,3,15,23] for a more detailed description of Up_One, Up_All and their applications.

3 Satisfiability of Formulas in CNF

A Boolean formula is in conjunctive normal form (on clausal form) if it is represented as a conjunction (AND) of clauses, each of which is the disjunction (OR) of one or more literals. A literal is either a variable or the negation of a variable. The Davis-Putnam algorithm [11,12] (see Algorithm 1) determines whether a Boolean formula ϕ in CNF is satisfiable. Line 1 is the base case where ϕ is the empty set of clauses which represents "true." Line 3 is the backtracking step where ϕ contains an empty clause which represents "false." Line 5 handles unit

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 $choose\text{-}literal(\phi)$ selects a literal for DP to split on.

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

clauses, i.e., clauses of the form x or $\neg x$. In this case, the value of the variable in the unit clause l is assigned in all remaining clauses of ϕ using the $assign(l, \phi)$ procedure. Line 8 and 9 handles the general case where a literal is chosen and the algorithm splits on whether the literal is true or false. There are a number of different heuristics for choosing a "good" literal in line 8 and the SAT-solvers based on Davis-Putnam differ by how they choose the literals to split on. A simple heuristic is to choose the literal in such a way that the assignments in line 9 produce the most unit clauses.

4 Satisfiability of Non-clausal Formulas

Using BEDs, the effect of splitting on a literal is obtained 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 contains the variable x. Both *low* and *high* children are BEDs. The formula is satisfiable if either the *low* child or the *high* child (or both) represents 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.

This suggests a recursive algorithm that pulls variables up one at a time. If the algorithm at any point reaches the terminal **1**, then a satisfying assignment has been found (the path from the root to the terminal **1** gives the assignment). The test for the empty set of clauses (line 1 in Algorithm 1) 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**. It's not possible to test for unit clauses in the BED and thus lines 5 and 6 have no correspondence in the BED algorithm. The only use of the unit clause detection in the Davis-Putnam algorithm is to reduce the size of the CNF representation. However, the BED data structure has a large number of built-in reduction rules such as the distributive laws and the absorption laws [23].

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

```
Name: BEDSAT u
 1: if u = 1 then
 2:   return true
 3: else if u = 0 then
 4:   return false
 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'
```

These reduction rules are applied each time a new BED vertex is created and can potentially reduce the size of the representation considerably. Algorithm 2 shows the pseudo-code for the SAT-procedure BEDSAT.

The function *choose-variable* in line 6 of Algorithm 2 selects a variable to split on. With a clausal 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. Although we do not have a clausal representation of the formulas when using BEDs, it is still posible to choose good candidate variables. In [23], several different heuristics for picking good variable orderings for UP_ONE are discussed. The first variable in such an ordering is probably a good variable to split on. In the prototype implementation, a simple strategy has been implemented: the first variable encountered during a depth-first search is used in the spilt. Notice that we do not need to split on the variables in the same order along different branches, i.e., it is not necessary to choose a single global variable ordering. Thus, the variable ordering can be adjusted to each sub-BED as the algorithm executes.

In line 9 the algorithm branches out in two: one branch for the low child and one for the high child. If a satisfying assignment is found in one branch (and thus BEDSAT returns true), it is not necessary to consider the other branch. We have implemented a simple greedy strategy of first examining the branch with the smaller BED size (least number of vertices).

An interesting feature of the BEDSAT algorithm is that it is possible to compute the fraction of the state space that has been examined at any point in the execution. It is known that the algorithm will terminate when 100% of the state space has been examined (it may of course terminate earlier if a satisfying assignment is found.) Figure 1 shows graphically how to determine the fraction of the state space that has been examined by the BEDSAT algorithm. 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

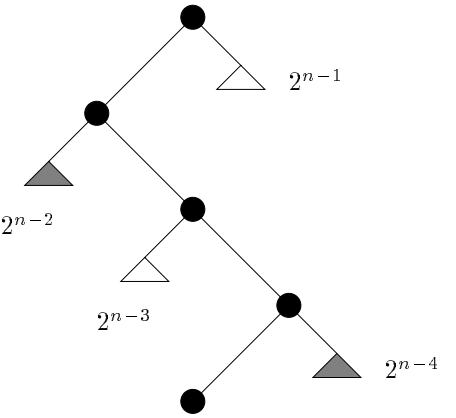

Fig. 1. Illustration of how to determine the percentage of the state space that has been examined. Each circle represents a split on a variable. The top circle is the starting point. The triangles represent sub-BEDs; the white ones are as yet unexamined while the gray ones have already been examined. 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 $\frac{2^{n-2}+2^{n-4}}{2^n}$.

each triangle assuming that there are n variables in total. The fraction of the state space examined so far is determined by adding the numbers from the gray triangles and dividing by 2^n which is the size of the complete state space.

Of course, the percentage of the state space that has been examined does not say much about the time remaining in the computation. However, it does allow us to detect whether the algorithm is making progress. One could imagine a 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. This could also be done automatically by tracking how the percentage changes over time. No or little growth could indicate that the choosen sequence of variables to split on is inefficient and the algorithm should backtrack and pick new split variables. Such backtracking is called *premature* and the technique is used in many implementations of Davis-Putnam. It works the best for satisfiable functions since it allows the search to give up on a particular part of the state space and concentrate on other, and hopefully easier, parts. If a satisfying assignment is found in the easy part of the state space, then the difficult part never needs to be revisited. For unsatisfiable functions, the entire state space needs to be examined, and giving up on one part just postpones the problems. The only hope is that by choosing a different sequence of variables to split on, the BED reduction rules collapse the difficult part of the state space.

The BEDSAT algorithm can be improved by combining it with traditional BDD construction. As more and more splits are performed, more and more

Algorithm 3 The BEDSAT algorithm with cutoff size N. $|u|$ is the number of vertices in the BED u. Line 6 returns whether the BED u represents a satisfiable function.

Name: BEDSAT u
1: **if** $u = \mathbf{1}$ **then**
2: **return** true
3: **else if** $u = \mathbf{0}$ **then**
4: **return** false
5: **else if** $|u| < N$ **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'

variables are assigned a value and the remaining BED shrinks. The BEDSAT algorithm, as described above, continues this process until either reaching a terminal **1**, or the entire BED is reduced to **0** (i.e., the BED is unsatisfiable). However, at some point it becomes more efficient to convert the BED into a BDD from which it can be decided immediately whether the original formula is satisfiable (the BDD is *not* **0**) or the algorithm has to backtrack and continue spliting (the BDD is **0**).

As discussed in the introduction, there is a trade-off between building the BDD and splitting on variables. The BDD construction computes too much information and is slow for large BEDs. On the other hand, splitting on variables tend to be slow when the depth is large. To be able to find the optimal point between the BDD construction and splitting on variables, we use a *cutoff size* N for the remaining BED; see Algorithm 3. If the size of the BED of a subproblem less than the cutoff size, the BED is converted into a BDD, otherwise we continue splitting. For large values of N, the revised version of BEDSAT reduces to pure BDD construction. For N equal to 0, the revised algorithm is identical to Algorithm 2.

5 Experimental Results

To see how well BEDSAT works in practice, we compare it to other techniques for solving satisfiability problems. Unfortunately, the standard benchmarks used to evaluate SAT-solvers are all in CNF (see for example [14]). To compare the performance of BEDSAT with existing algorithm on non-clausal Boolean formulas, we obtain Boolean formulas from the circuits in the ISCAS'85 benchmark siute [8] and from model checking [24].

We compare BEDSAT to the BDD construction algorithms UP_ONE and UP_ALL, both using the FANIN variabel ordering heuristic. Furthermore, we

compare BedSat with the state-of-the-art SAT-solvers Sato and Grasp. Since both Sato and Grasp require their input to be in CNF form, we convert the BEDs to CNF although this increases the number of variables and thus also the state space for Sato and Grasp.

The experiments are performed on a 450 MHz Pentium III PC running Linux. For the BDD construction in Algorithm 3 we use Up_All with the Fanin variable ordering heuristic [23]. All runs are limited to 32 MB of memory and 15 minutes of CPU time.

Table 1 shows the ISCAS'85 results. The ISCAS'85 benchmark consists of eleven multi-level combinational circuits, nine of which exist both in a redundant and a non-redundant version. Furthermore, the benchmark contains five circuits that originally were believed to be non-redundant versions but it turned out that they contained errors and weren't functionally equivalent to the original circuits [16]. The nine equivalent pairs of circuits corresponds to 475 unsatisfiable Boolean formulas (the circuits have several outputs) when the outputs of the circuits are pairwise exclusive-or'ed. The first nine rows of Table 1 show the runtimes to prove that the 475 fomulas are unsatisfiable using Up_One, Up_All, Sato, Grasp, and the BedSat algorithm with the cutoff size equal to 0, 100, 400 and 1000. Up_One and Up_All perform quite well on all nine circuits. 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 several 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 formulas. Increasing the cutoff size to 100 or 400 improves 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 1 show the results for the erroneous circuits. Here there are 340 Boolean formulas 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 considerably 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 size of 0. In the correct version of the circuits, BedSat uses 185 seconds. This number reduces to 35.9 seconds for the erroneous circuits. The `c3540` circuit 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 are spent on proving those five outputs to be unsatisfiable. Another example is `c1908` where, in the correct case, BedSat spends all the time (242 seconds) on one

Table 1. 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, 400 and 1000 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			
						0	100	400	1000
c432/nr	U	2.1	1.7	0.5	0.4	36.4	3.5	1.4	1.4
c499/nr	U	4.3	1.8	1.8	1.4	17.8	16.7	1.7	1.7
c1355/nr	U	4.3	1.8	1.8	1.5	18.1	16.5	1.7	1.7
c1908/nr	U	0.7	0.6	0.4	0.4	242	11.1	0.2	0.2
c2670/nr	U	1.2	0.6	1.0	0.9	38.6	1.9	0.3	0.3
c3540/nr	U	32.3	39.2	–	–	185	133	10.9	16.3
c5315/nr	U	16.2	1.9	–	15.0	1.1	1.0	0.9	1.3
c6288/nr	U	2.7	–	–	–	–	–	–	–
c7552/nr	U	3.6	1.1	–	4.4	–	–	0.7	0.7
c1908/nr−err	S/U	0.7	0.6	0.4	0.4	0.1	0.1	0.2	0.2
c2670/nr−err	S/U	2.9	0.7	0.9	0.8	0.4	0.3	0.3	0.3
c3540/nr−err	S/U	42.8	40.2	–	–	35.9	15.8	4.6	6.5
c5315/nr−err	S/U	32.7	2.4	31.7	10.3	0.7	1.6	1.5	1.8
c7552/nr−err	S/U	8.1	1.8	2.5	2.6	176	2.0	1.3	1.3

unsatisfiable output. In the erroneous version the difficult output has an error and the corresponding Boolean formula becomes satisfiable. BEDSAT finds a satisfying assignment instantaneously (0.1 seconds).

By varying the cutoff size, we can control whether BEDSAT works mostly as the standard BDD construction (when using a high cutoff size) or as a Davis-Putnam SAT-solver (a low cutoff size). From Table 1 it is observed that for the ISCAS circuits, a cutoff size of 400 seems to be the optimal value. Using this value for the cutoff size, the BEDSAT algorithm outperforms both pure BDD construction (UP_ALL and UP_ONE) and standard SAT-solvers (SATO, GRASP, and BEDSAT with 0 cutoff) for all the larger circuits except `c6288/nr` and for all the errorneous (and thus satisfiable) ISCAS circuits.

The Boolean formulas obtained from the ISCAS'85 circuits have many identical sub-formulas since they are obtained by comparing two similar circuits. To test the BEDSAT algorithm on a more reaslistic example (at least from the point of view of formal verification), we have extracted Boolean formulas that arise during the fixed-point iteration when performing model checking of a 16-bit shift-and-add multiplier [23]. Table 2 shows the results for the model checking problems. The numbers 10, 20 and 30 indicate the output bit we are considering. The word "final" indicates the satisfiability problem for the check for whether the

Table 2. 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.

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

implementation satisfies the specification. The word "last_fp" indicates the satisfiability problem for the last iteration in the fixed-point computation (where it is detected that the fixed-point is reached). The word "second_last_fp" indicates the satisfiability problem for the previous iteration in the fixed-point iteration. 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 is more consistent in its performance.

6 Conclusion

This paper has presented the BEDSAT algorithm for solving the satisfiability problem on BEDs. The algorithm adopts the Davis-Putnam algorithm to the

BED data structure. Traditional SAT-solvers require the Boolean formula to be given in CNF, but BedSat works directly on the BED and thus avoids the conversion of the formula to CNF which either adds extra variables or may result in an exponentially larger CNF formula. The BedSat algorithm is also able to take advantage of the BED data structure by using the reduction rules from [23] during the algorithm and by taking advantage of the sharing of sub-formulas.

We have described how the BedSat algorithm is combined with traditional BDD construction. By adjusting a single parameter to the BedSat algorithm it is possible to control to what extent the algorithm behaves like the Apply-algorithm or like a SAT-solver. Thus the algorithm can be seen as bridging the gap between standard SAT-solvers and BDDs.

We present promising experimental results for 566 non-clausal formulas obtained from the multi-level combinational circuits in the ISCAS'85 benchmark suite and from performing bounded model checking of a shift-and-add multiplier. For these formulas, the BedSat algorithm is more efficient than both pure SAT-solvers (Sato and Grasp) and standard BDD construction. The combination works especially well on formulas 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. Holger H. Hoos and Thomas Sttzle. SATLIB – the satisfiability library. `http://aida.intellektik.informatik.tu-darmstadt.de/~hoos/SATLIB/`.
15. 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.
16. W. Kunz and D. K. Pradhan. Recursive learning: A new implication technique for efficient solutions to CAD problems – test, verification, and optimization. *IEEE Transactions on Computer Aided Design*, 13(9):1143–1158, September 1994.
17. J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48, 1999.
18. R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 42–47, 1993.
19. R. Sebastiani. Applying GSAT to non-clausal formulas. *Journal of Artificial Intelligence Research (JAIR)*, 1:309–314, January 1994.
20. 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.
21. 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.
22. 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.
23. 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.
24. 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.
25. 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.