Extended Resolution Proofs for Conjoining BDDs

Carsten Sinz and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University Linz, Austria
{carsten.sinz, armin.biere}@jku.at

Abstract. We present a method to convert the construction of binary decision diagrams (BDDs) into extended resolution proofs. Besides in proof checking, proofs are fundamental to many applications and our results allow the use of BDDs instead—or in combination with—established proof generation techniques, based for instance on clause learning. We have implemented a proof generator for propositional logic formulae in conjunctive normal form, called EBDDRES. We present details of our implementation and also report on experimental results. To our knowledge this is the first step towards a practical application of extended resolution.

1 Introduction

Propositional logic decision procedures [1–6] lie at the heart of many applications in hard- and software verification, artificial intelligence and automatic theorem proving [7–12]. They have been used to successfully solve problems of considerable size. In many practical applications, however, it is not sufficient to obtain a yes/no answer from the decision procedure. Either a model, representing a sample solution, or a justification, why the formula possesses none is required. So, e.g. in declarative modeling or product configuration [9, 10] an inconsistent specification given by a customer corresponds to an unsatisfiable problem instance. To guide the customer in correcting his specification a justification why it was erroneous can be of great help. In the context of model checking proofs are used, e.g., for abstraction refinement [11], or approximative image computations through interpolants [13]. In general, proofs are also important for certification through proof checking [14].

Using BDDs for SAT is an active research area [15–20]. It turns out that BDD [21] and search based techniques [2] are complementary [22, 23]. There are instances for which one works better than the other. Therefore, combinations have been proposed [16, 17, 20] to obtain the benefits of both, usually in the form of using BDDs for preprocessing. However, in all these approaches where BDDs have been used, proof generation has not been possible so far.

In our approach, conjunction is the only BDD operation considered to date. Therefore our solver is far less powerful than more sophisticated BDD-based SAT solvers [15–20]. In particular, we currently cannot handle existential quantification. However, our focus is on proof generation, which none of the other approaches currently supports. We also conjecture that similar ideas as presented in this paper can be used to produce proofs for BDD-based existential quantification and other BDD operations. This will eventually allow us to generate proofs for all the mentioned approaches.

We have chosen extended resolution as a formalism to express our proofs, as it is on the one hand a very powerful proof system equivalent in strength to extended Frege systems [24], and on the other hand similar to the well-known resolution calculus [25]. Despite its strength, it still offers simple proof checking: after adding a check to avoid cyclical definitions, an ordinary proof checker for resolution can be used.

Starting with [26], extended resolution has been mainly a subject of theoretical studies [27, 24]. In practical applications it did not play an important role so far. This may be due to the fact that direct generation of (short) extended resolution proofs is very hard, as there is not much guidance on how to use the *extension rule*. However, when "proofs" are generated by another means (by BDD computations in our case), extended resolution turns out to be a convenient formalism to concisely express proofs. We expect that a wide spectrum of different propositional decision procedures can be integrated into a common proof language and proof verification system using extended resolution.

The rest of this paper is organized as follows: First, we give short introductions to extended resolution and BDDs. Then we present our method to construct extended resolution proofs out of BDD constructions. Thereafter, we portray details of our implementation EBDDRES and show experimental results obtained with it. Finally, we conclude and give possible directions for future work.

2 Theoretical Background

In this paper we are mainly dealing with propositional logic formulae in conjunctive normal form (CNF). A formula F (over a set of variables V) in CNF is a conjunction of clauses, where each clause is a disjunction of literals. A literal is either a variable or its negation. We use capital letters $(C, D, \dots)$ to denote clauses and small-case letters to denote variables $(a, b, c, \dots x, y, \dots)$ and literals $(l, l_1, l_2, \dots)$. Instead of writing a clause $C = (l_1 \vee \dots \vee l_k)$ as a disjunction, we alternatively write it as a set of literals, i.e. as $\{l_1, \dots, l_k\}$, or in an abbreviated form as $(l_1 \dots l_k)$. For a literal l, we write $\bar{l}$ for its complement, i.e $\bar{x} = \neg x$ and $\overline{\neg x} = x$ for a variable x.

2.1 Extended Resolution

Extended resolution (ER) was proposed by Tseitin [26] as an extension of the resolution calculus [25]. The resolution calculus consists of a single inference rule,[1]

$$\frac{C \dot{\cup} \{l\} \quad \{\bar{l}\} \dot{\cup} D}{C \cup D}$$

and is used to refute propositional logic formulae in CNF. Here C and D are arbitrary clauses and l is a literal. A refutation proof is achieved, when the empty clause (denoted by $\square$) can be derived by a series of resolution rule applications. Extended

[1] By $\dot{\cup}$ we denote the disjoint union operation, i.e. $A \dot{\cup} B$ is the same as $A \cup B$ with the additional restriction that $A \cap B = \emptyset$.

resolution adds an *extension rule* to the resolution calculus, which allows introduction of definitions (in the form of additional clauses) and new (defined) variables into the proof. The additional clauses must stem out of the CNF conversion of definitions of the form $x \leftrightarrow F$, where F is an arbitrary formula and x is a new variable, i.e. a variable neither occurring in the formula we want to refute or in previous definitions nor in F. In this paper—besides introducing two variables for the Boolean constants—we only define new variables for if-then-else (*ITE*) constructs written as $x\ ?\ a : b$ (for variables x, a, b), which is an abbreviation for $(x \rightarrow a) \wedge (\neg x \rightarrow b)$. So introduction of a new variable w as an abbreviation for $ITE(x, a, b) = x\ ?\ a : b$ is reflected by the rule

$$\frac{}{(\bar{w}\bar{x}a)(\bar{w}xb)(w\bar{x}\bar{a})(wx\bar{b})}$$

which has no premises, and introduces four new clauses at once for the given instance of the *ITE* construct. We have used the abbreviated notation for clauses here that leaves out disjunction symbols. Concatenation of clauses is assumed to denote the conjunction of these. It should also be noted that the extended clause set produced by applications of the extension rule is only equisatisfiable to the original clause set, but not equivalent.

The interest in extended resolution stems from the fact that no super-polynomial lower bound is known for extended resolution [24] and that it is comparable in strength to the most powerful proof systems (extended Frege) for propositional logic [24]. This also means that for formulae which are hard for resolution (e.g. Haken's pigeon-hole formulae [28]) short ER proofs exist [27]. Moreover, as it is an extension of the resolution calculus, ER is a natural candidate for a common proof system integrating different propositional decision procedures like the resolution-based DPLL algorithm [1, 2] .

2.2 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) were proposed by Bryant [21] to compactly represent Boolean functions as DAGs (directed acyclic graphs). In their most common form as reduced ordered BDDs (that we also adhere to in this paper) they offer the advantage that each Boolean function is uniquely represented by a BDD, and thus all semantically equivalent formulae share the same BDD. BDDs are based on the Shannon expansion

$$f = ITE(x, f_1, f_0) = (x \rightarrow f_1) \wedge (\neg x \rightarrow f_0)\ ,$$

decomposing f into its *co-factors* f_0 and f_1 (w.r.t variable x). The co-factor f_0 (resp. f_1) is obtained by setting variable x to false (resp. true) in formula f and subsequent simplification. By repeatedly applying Shannon expansion to a formula selecting splitting variables according to a global variable ordering until no more variables are left (ending in terminal nodes 0 and 1), its BDD representation is obtained (resembling a decision tree). Merging equivalent nodes (i.e. same variable and co-factors) and deleting nodes with coinciding co-factors results in reduced ordered BDDs. Fig. 1 shows the BDD representation of formula $f = x \vee (y \wedge \neg z)$.

To generate BDDs for (complex) formulae, instead of performing Shannon decomposition and building them *top-down*, they are typically built *bottom-up* starting with basic BDDs for variables or literals, and then constructing more complex BDDs by

Algorithm 1: BDD-and(a, b)

```
1: if $a = 0$ or $b = 0$ then return $0$
2: if $a = 1$ then return $b$ else if $b = 1$ then return $a$
3: $(x, a_0, a_1)$ = decompose(a); $(y, b_0, b_1)$ = decompose(b)
4: if $x < y$ then return new-node($y$, BDD-and($a, b_0$), BDD-and($a, b_1$))
5: if $x = y$ then return new-node($x$, BDD-and($a_0, b_0$), BDD-and($a_1, b_1$))
6: if $x > y$ then return new-node($x$, BDD-and($a_0, b$), BDD-and($a_1, b$))
```

using BDD operations (e.g., *BDD-and*, *BDD-or*) for logical connectives. As we will need it in due course, we give the *BDD-and* algorithm explicitly (Algorithm 1). Here, *decompose* breaks down a non-terminal BDD node into its constituent components, i.e. its variable and cofactors. The function *new-node* constructs a new BDD node if it is not already present, and otherwise returns the already existent node. The comparisons in steps 4 to 6 are based on the global BDD variable order.

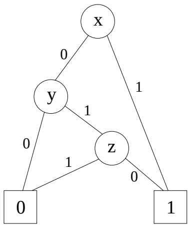

Fig. 1. BDD representation of formula $x \vee (y \wedge \neg z)$ using variable ordering $x > y > z$.

3 Proof Construction

We assume that we are given a formula F in CNF for which we want to construct an ER proof that shows unsatisfiability of F (i.e., we show that $\neg F$ is a tautology). Instead of trying to derive such a proof directly in the ER calculus—which could be quite hard, as there are myriads of ways to introduce new definitions—we first construct a BDD equivalent to formula F and then extract an ER proof out of this BDD construction. The BDD for formula F is built gradually (bottom-up) by conjunctively adding more and more clauses to an initial BDD representing the Boolean constant true.

Starting with an ordered set of clauses $S = (C_1, \ldots, C_m)$ for formula $F_S = C_1 \wedge \cdots \wedge C_m$ (which we want to proof unsatisfiable), we thus first build a BDD c_i for each clause C_i. Then we construct intermediate BDDs h_i corresponding to partial conjunctions $C_1 \wedge \cdots \wedge C_i$, until, by computing h_m, we have reached a BDD for the whole formula. These intermediate BDDs can be computed recursively by the equations

$$h_2 \leftrightarrow c_1 \wedge c_2 \qquad \text{and} \qquad h_i \leftrightarrow h_{i-1} \wedge c_i \quad \text{for } 3 \leq i \leq m \ .$$

If h_m is the BDD consisting only of the 0-node, we know that formula F_S is unsatisfiable and we can start building an ER proof.

The method to construct the ER proof works by first introducing new propositional variables (thus ER is required), one for each node of each BDD that occurs during the construction process, i.e. for all c_i and h_i. New variables are introduced based on the Shannon expansion of a BDD node: for an internal node f containing variable x and having child nodes f_1 and f_0, a new variable (which we also call f), defined by

$$f \;\leftrightarrow\; (x \mathrel{?} f_1 : f_0) \qquad\qquad (\bar f \bar x f_1)(\bar f x f_0)(f \bar x \bar f_1)(f x \bar f_0)$$

is introduced. We have also given the clausal representation of the definition on the right. Terminal nodes are represented by additional variables n_0 and n_1 defined by

$$n_0 \;\leftrightarrow\; 0 \qquad (\bar n_0) \qquad\quad \text{and} \qquad\quad n_1 \;\leftrightarrow\; 1 \qquad (n_1)\ .$$

Note that introducing new variables in this way does not produce cyclic definitions, as the BDDs themselves are acyclic. So by introducing variables bottom-up from the leaves of the BDD up to the top node, we have an admissible ordering for applying the extension rule.

With these definitions, we can give an outline of the ER proof we want to generate. It consists of three parts: first, we derive unit clauses (c_i) for the variables corresponding to the top BDD nodes of each clause. Then out of the recursive runs of each *BDD-and*-operation we build proofs for the conjunctions $h_{i-1} \wedge c_i \leftrightarrow h_i$ (in fact, only the implication from left to right is required). And finally, we combine these parts into a proof for h_m. If h_m is the variable representing the zero node, i.e. $h_m = n_0$, we can derive the empty clause by another single resolution step with the defining clause for n_0. We thus have to generate ER proofs for all of the following:

$$S \vdash c_i \qquad\qquad \text{for all } 1 \le i \le m \tag{ER-1}$$

$$S \vdash c_1 \wedge c_2 \rightarrow h_2 \tag{ER-2a}$$

$$S \vdash h_{i-1} \wedge c_i \rightarrow h_i \qquad\qquad \text{for all } 3 \le i \le m \tag{ER-2b}$$

$$S \vdash h_m \tag{ER-3}$$

For a proof of (ER-1) for some i assume that clause $D = C_i$ consists of the literals $(l_1, \dots, l_k)$. We assume literals to be ordered decreasingly according to the BDD's global variable ordering. Then the newly introduced variables for the nodes of the BDD representation of clause D are $d_j \leftrightarrow (l_j \mathrel{?} n_1 : d_{j+1})$ if l_j is positive, and $d_j \leftrightarrow (\bar l_j \mathrel{?} d_{j+1} : n_1)$ if l_j is negative. We identify d_{k+1} with n_0, and c_i with d_1 here. These definitions induce—among others—the clauses $(d_j \bar l_j \bar n_1)$ and $(d_j l_j \bar d_{j+1})$ for all $1 \le i \le k$. We therefore obtain the following ER proof for (d_1): First, we derive $(d_k l_1 \dots l_{k-1} \bar n_1)$ by resolving $(l_1 \dots l_k)$ with $(d_k \bar l_k \bar n_1)$. Then, iteratively for $j = k$ down to $j = 2$, we derive $(d_{j-1} l_1 \dots l_{j-2} \bar n_1)$ from $(d_j l_1 \dots l_{j-1} \bar n_1)$ by

$$\dfrac{\dfrac{(d_j l_1 \dots l_{j-1} \bar n_1) \quad (d_{j-1} l_{j-1} \bar d_j)}{(d_{j-1} l_1 \dots l_{j-1} \bar n_1)} \qquad (d_{j-1} \bar l_{j-1} \bar n_1)}{(d_{j-1} l_1 \dots l_{j-2} \bar n_1)}$$

And finally, we derive the desired (d_1) by resolving $(d_1\bar{n}_1)$ with (n_1).

For (ER-2a) and (ER-2b), assume that we want to generate a proof for the general implication $f \wedge g \rightarrow h$ (i.e. for the clause $(\bar{f}\bar{g}h)$), where we have h computed as the conjunction of f and g by the *BDD-and* operation. Definitions for the BDD nodes resp. variables are given by the Shannon expansion of f, g and h as

$$\begin{array}{rclcl} f & \leftrightarrow & (x \mathrel{?} f_1 : f_0) & \qquad & (\bar{f}\bar{x}f_1)(\bar{f}xf_0)(f\bar{x}\bar{f}_1)(fx\bar{f}_0) \\ g & \leftrightarrow & (x \mathrel{?} g_1 : g_0) & \qquad & (\bar{g}\bar{x}g_1)(\bar{g}xg_0)(g\bar{x}\bar{g}_1)(gx\bar{g}_0) \\ h & \leftrightarrow & (x \mathrel{?} h_1 : h_0) & \qquad & (\bar{h}\bar{x}h_1)(\bar{h}xh_0)(h\bar{x}\bar{h}_1)(hx\bar{h}_0) \ . \end{array}$$

We can now recursively construct an ER proof for the implication $f \wedge g \rightarrow h$, where in the recursion step we assume that proofs for both $f_0 \wedge g_0 \rightarrow h_0$ and $f_1 \wedge g_1 \rightarrow h_1$ are already given. We thus obtain:

$$\dfrac{\dfrac{(hx\bar{h}_0) \quad \dfrac{(\bar{g}xg_0) \quad \dfrac{(\bar{f}xf_0) \quad \dfrac{\vdots}{(\bar{f}_0\bar{g}_0h_0)}}{(\bar{f}x\bar{g}_0h_0)}}{(\bar{f}\bar{g}xh_0)}}{(\bar{f}\bar{g}hx)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\vdots}{(\bar{f}_1\bar{g}_1h_1)} \quad (\bar{f}\bar{x}f_1)}{(\bar{f}\bar{x}\bar{g}_1h_1)} \quad (\bar{g}\bar{x}g_1)}{(\bar{f}\bar{g}\bar{x}h_1)} \quad (h\bar{x}\bar{h}_1)}{(\bar{f}\bar{g}h\bar{x})}}{(\bar{f}\bar{g}h)}$$

The recursive process stops when we arrive at the leave nodes resp. the base case of the recursive *BDD-and* algorithm. This algorithm stops, if either of its arguments is a Boolean constant, or when both arguments are the same. In each case we obtain tautological clauses, like e.g. $(\bar{n}_0\bar{g}n_0)$ for f being the 0-BDD, or $(\bar{f}f)$ for computing the conjunction of two identical arguments. We call this a *trivial step* of the *BDD-and* algorithm. Moreover, we may stop the recursion, if f implies g (or vice versa), as we then have generated the tautological clause $(\bar{f}gf)$. This case we call a *redundant step* of the algorithm. We will show below how such tautological clauses can be avoided or eliminated in the ER proof.

For a proof of (ER-3) we just have to combine parts one and two: We can resolve the derived unit clauses (c_1) and (c_2) with the implication $(\bar{c}_1\bar{c}_2h_2)$ to produce a proof of (h_2), and then continue iteratively using (ER-2b) to derive further unit clauses (h_i) for all $3 \leq i \leq m$. Finally, (h_m) is resolved with $(\bar{n}_0)$ to produce the empty clause. This completes the whole ER proof for unsatisfiability of F_S.

3.1 Avoiding Tautological Clauses

The proofs (ER-2a) and (ER-2b) presented above may contain tautological clauses that are introduced by equivalent BDD nodes. So if, e.g., $f_0 = h_0$, the recursive proof step starts with a tautological clause in the left branch. We now give a detailed construction that avoids such clauses completely, starting with some definitions:

Definition 1 (Line, Step). *A triplet of BDD nodes (f, g, h) such that h is the BDD node representing $f \wedge g$ is called a (BDD-and) step. A step is called trivial if f or g are*

Boolean constants or if $f = g$. A non-trivial step is also called a (cache) line. Steps and lines are also identified with clauses, where a line (f, g, h) corresponds to the clause $(\bar{f}\bar{g}h)$. We identify nodes with ER variables here.

Definition 2 (Redundancy). *A line $L = (f, g, h)$ is called redundant if $f = h$ or $g = h$, otherwise it is called irredundant. The notion of redundancy also carries over to the clause $(\bar{f}\bar{g}h)$ corresponding to line L.*

When we have reached an irredundant step $(\bar{f}\bar{g}h)$, we can check whether the co-factor clauses of the assumptions $(\bar{f}_0\bar{g}_0h_0)$ and $(\bar{f}_1\bar{g}_1h_1)$ of the step are redundant. If this is the case, the proof has to be simplified and recursion stops (in all but one case) at the redundant step. We now give simplified proofs that contain no tautological clauses for the recursion step of the proofs (ER-2a) and (ER-2b). In what follows, we call the sub-proof of $(\bar{f}\bar{g}hx)$ out of $(\bar{f}_0\bar{g}_0h_0)$ the left branch and the sub-proof of $(\bar{f}\bar{g}h\bar{x})$ out of $(\bar{f}_1\bar{g}_1h_1)$ the right branch of the recursive proof step.

R1 If $f_0 = h_0$, we obtain a proof for the left branch (and analogously for $g_0 = h_0$ and for $f_1 = h_1$ or $g_1 = h_1$ on the right branch) by resolving $(hx\bar{h}_0)$ and $(\bar{f}xf_0)$ to produce $(\bar{f}hx)$. Although we have proved a stronger claim on the left branch in this case, it cannot happen that g also disappears on the right branch, as this would only be possible if $f_1 = h_1$. But then $f = h$ would also hold and the step $(\bar{f}\bar{g}h)$ would already be redundant, contradicting our assumption.

T1 If $f_0 = g_0$ (this is not a tautological case, however) then $h_0 = f_0 = g_0$ also holds, so that we arrive at the case above (and similarly for $f_1 = g_1$). We can even choose which of the definitions (either for f or for g) we want to use.

T2 If $f_0 = 1$ we obtain $h_0 = g_0$ and we can use the proof given under (R1) for the left branch (similar for $g_0 = 1$, $f_1 = 1$, and $g_1 = 1$). If $f_0 = 0$, we can use the definition $(\bar{f}xn_0)$ of f and $(\bar{n}_0)$ of 0 to derive the stronger $(\bar{f}x)$. It cannot happen that $f_1 = 0$ at the same time (as then the step would be trivial), so the only possibility where we are really left with a stronger clause than the desired $(\bar{f}\bar{g}h)$ occurs when $f_0 = 0$ and $g_1 = 0$ (or $f_1 = g_0 = 0$). Then we have $h = 0$ and we can derive $(\bar{f}\bar{g})$. In this case we just proceed as in case (H0) below.

H0 If $h = 0$ we let $h_0 = h_1 = 0$ and recursively generate sub-proofs skipping the definition of h by rule (X1) below.

H1 The case $h = 1$ could only happen if $f = g = 1$ would also hold. But then the step would be redundant. If $h_0 = 1$ we derive the stronger (hx) by resolving $(hx\bar{n}_1)$ with (n_1), and similar for $h_1 = 1$. It cannot happen that we have $h_0 = h_1 = 1$ at the same time, as this would imply $h = 1$. Thus, on the other branch we always obtain a clause including $\bar{f}$ and $\bar{g}$ and therefore the finally resulting clause is always $(\bar{f}\bar{g}h)$.

X1 If the decision variable x does not occur in one or several of the BDDs f, g, or h (i.e., for example, if $f = f_0 = f_1$) the respective resolution step(s) involving $\bar{f}$, $\bar{g}$, or h, can just be skipped.

Note that in all degenerate cases besides cases (T2) (only for $f_0 = g_1 = 0$ or $f_1 = g_0 = 0$) and (H0) the proof stops immediately and no recursive descent towards the leaves of the BDD is necessary. If the last proof step in any of the sub-proofs of (ER-2a) or (ER-2b) results in a redundant step, the proof of part (ER-3) contains tautological clauses and must be simplified. So assume in (ER-2a) that the last step is redundant, i.e. in $(\bar{c}_1\bar{c}_2h_2)$ either $c_1 = h_2$ or $c_2 = h_2$ holds. In both cases the *BDD-and* computation is not needed and we can skip the resolution proof for h_2 and use clause c_1 or c_2, whichever is equivalent to h_2, in the subsequent proof. The same holds for (ER-2b), if the last proof step is redundant and we thus have $h_{i-1} = h_i$ or $c_i = h_i$. Again, we can drop the *BDD-and* computation and skip the resolution proof for h_i out of h_{i-1} and c_i, but instead using h_{i-1} or c_i directly.

A further reduction in proof size could be achieved by simplifying definitions introduced for BDD nodes with constants 0 and 1 as co-factors. So instead of using $f \leftrightarrow (x \ ? \ n_1 : f_0)$ as definition for a node f with constant 1 as its first co-factor, we could use the simplified definition $f \leftrightarrow (\bar{x} \rightarrow f_0)$ which would result in only three simpler clauses $(\bar{f}xf_0)$, $(f\bar{x})$, and $(f\bar{f}_0)$.

4 Implementation and Experimental Results

We have implemented our approach in the SAT solver EBDDRES. It takes as input a CNF in DIMACS format and computes the conjunction of the clauses after transforming them into BDDs. The result is either the constant zero BDD, in which case the formula is unsatisfiable, or a non-zero BDD. In the latter case a satisfying assignment is generated by traversing a path from the root of the final BDD to the constant one leaf.

In addition to solving the SAT problem, a proof trace can be generated. If the formula is unsatisfiable the empty clause is derived. Otherwise a unit clause can be deduced. It contains a single variable which represents the root node of the final BDD. The trace format is similar to the trace format used for ZCHAFF [14] or MINISAT [6]. In particular, we do not dump individual resolution steps, but combine piecewise regular input resolution steps into *chains*, called *trivial* resolution steps in [29]. Each chain has a set of *antecedent* clauses and one *resolvent*. The antecedents are treated as input clauses in the regular input resolution proof of the resolvent. Our trace checker is able to infer the correct resolution order for the antecedents by unit propagation after assuming the negation of the resolvent clause. Original clauses and those used for defining new variables in the extended resolution proof are marked by an empty list of antecedents. Note that a proof checker for the ordinary resolution calculus is sufficient for extended resolution proofs, too, as all definitional clauses produced by the extension rule can be added initially, and then only applications of the resolution rule are required to be checked.

The ASCII version of the trace format itself is almost identical to the DIMACS format and since the traces generated by EBDDRES are quite large we also have a compact binary version, comparable to the one used by MINISAT. Currently the translation from the default ASCII format into the binary format is only possible through an external tool. Due to this current limitation we were not able to generate binary traces where the ASCII trace was of size 1GB or more.

For the experiments we used a cluster of Pentium IV 3.0 GHz PCs with 2GB main memory running Debian Sarge Linux. The time limit was set to 1000 seconds, the mem-

Table 1. Comparison of trace generation with MINISAT and with EBDDRES.

	MINISAT			EBDDRES											
	solve		trace	solve		trace	trace size		bdd	recursive bdd and-steps					trace
	resources		size	resources		gen.	ASCII	binary	nodes	all	triv.	lines	red.	core	chk
	sec	MB	MB	sec	MB	sec	MB	MB	$\times 10^3$	$\times 10^3$	$\times 10^3$	$\times 10^3$	$\times 10^3$	$\times 10^3$	sec
ph7	0	0	0	0	0	0	1	0	3	20	10	10	0	10	0
ph8	0	4	1	0	3	0	3	1	15	67	34	33	0	33	0
ph9	6	4	11	0	3	0	3	1	8	90	45	45	0	45	0
ph10	44	4	63	1	17	1	30	10	136	538	270	269	1	268	2
ph11	887	6	929	1	13	1	21	8	35	670	335	334	1	333	2
ph12	*	-	-	2	28	1	33	12	31	1150	575	574	1	573	3
ph13	*	-	-	10	102	8	260	92	850	5230	2615	2614	2	2612	20
ph14	*	-	-	10	111	7	204	74	166	6554	3278	3276	2	3274	18
mutcb8	0	0	0	0	4	0	4	1	23	73	37	37	0	36	0
mutcb9	0	4	0	0	5	0	12	4	64	193	97	96	0	96	1
mutcb10	0	4	1	1	17	1	35	12	177	577	289	288	1	287	3
mutcb11	1	4	4	3	32	2	89	29	419	1380	691	690	3	686	6
mutcb12	8	4	22	6	62	5	188	64	906	2743	1372	1371	3	1368	13
mutcb13	113	5	244	15	146	12	452	155	2040	6398	3199	3198	8	3190	30
mutcb14	491	8	972	50	578	38	1465	*	6225	20520	10261	10260	20	10240	*
mutcb15	*	-	-	-	*	-	-	-	-	-	-	-	-	-	-
mutcb16	*	-	-	-	*	-	-	-	-	-	-	-	-	-	-
urq35	96	4	218	2	28	1	37	13	24	1216	608	608	0	608	3
urq45	*	-	-	-	*	-	-	-	-	-	-	-	-	-	-
fpga108	0	0		6	47	4	135	47	186	4087	2044	2043	3	2040	11
fpga109	0	0		3	44	2	70	24	83	2218	1109	1109	1	1108	6
fpga1211	0	0		54	874	38	1214	*	1312	33783	16892	16891	41	16850	*
add16	0	0	0	0	4	0	6	2	30	100	51	50	1	49	0
add32	0	0	0	1	9	1	24	8	122	445	223	222	4	217	2
add64	0	4	0	12	146	9	338	112	1393	5892	2948	2944	19	2925	23
add128	0	4	0	-	*	-	-	-	-	-	-	-	-	-	-

The first column lists the name of the instance. Columns 2-4 contain the data for MINISAT, first the time taken to solve the instance including the time to produce the trace, then the memory used, and in column 4 the size of the generated trace. The data for EBDDRES takes up the rest of the table. It is split into a more general part in columns 5-9 on the left. The right part provides more detailed statistics in columns 10-15. The first column in the general part of EBDDRES shows the time taken to solve the instance with EBDDRES including the time to generate and dump the trace. The latter is shown separately in column 7. The memory used by EBDDRES, column 6, is linearly related to the number of generated BDD nodes in column 10 and the number of generated cache lines in column 13. The number of recursive steps of the *BDD-and* operation occurs in column 11. Among these steps many trivial base cases occur (column 12) and the number of cache lines in column 13 is simply the number of non trivial steps. Among the cache lines several redundant lines occur (column 14) in which the result is equal to one of the arguments. The core consists of irredundant cache lines necessary for the proof. Their number is listed in the next to last column (column 15). The last column (column 16) shows the time taken by the trace checker to validate the proof generated by EBDDRES. The * denotes either *time out* (>1000 seconds) or *out of memory* (>1GB main memory).

ory limit to 1GB main memory and no size limit on the generated traces was imposed. Besides the *pigeon hole* instances (ph*) we used combinatorial benchmarks from [22, 30], more specifically *mutilated checker board* (mutcb*) and *Urquhart* formulae (urq*), FPGA routing (fpga*), and one suite of structural instances from [31], which represent equivalence checking problems for adder circuits (add*). The latter suite of benchmarks is supposed to be very easy for BDDs with variable quantification, which is not implemented in EBDDRES. Starting with ZCHAFF [4], these benchmarks also became easy for search based solvers.

As expected, the experimental data in Tab. 1 shows that even our simplistic approach in conjoining BDDs to solve SAT, is able to outperform MINISAT on certain hard combinatorial instances in the ph* and mutcb* family. In contrast to the simplified exposition in Sec. 3 a tree shaped computation turns out to be more efficient for almost all benchmarks. Only for one benchmark family (mutcb*) we used the linear combination. The results of Sec. 3 transfer to the more general case easily.

For comparison we used the latest version 1.14 of MINISAT, with proof generation capabilities. MINISAT in essence was the fastest SAT solver in the SAT'05 SAT solver competition. MINISAT in combination with the preprocessor SATELITE [32] was even faster in the competition, but we could not use SATELITE, because it cannot generate proofs. The binary trace format of EBDDRES (see column 9 of Tab. 1) is comparable—though not identical—to the trace format of MINISAT. EBDDRES was able to produce smaller traces for certain instances. Since for EBDDRES the traces grow almost linear in the number of steps, we expect much smaller traces for more sophisticated BDD-based SAT approaches.

Similar to related approaches, EBDDRES does not use dynamic variable ordering but relies on the choice of a good initial static order instead. We experimented with various static ordering algorithms. Only for the add* family of benchmarks it turns out that the variable order generated by our implementation of the FORCE algorithm of [33] yields better results. For all other families we used the original ordering. It is given by the order of the variable indices in the DIMACS file. Improvements on running times and trace size will most likely not be possible from better variable ordering algorithms. But we expect an improvement through clustering of BDDs and elimination of variables through existential quantification.

In general, whether BDD-based methods or SAT solvers behave superior turned out to be highly problem dependent, as other empirical studies also suggest [18, 19, 22].

5 Conclusion and Future Work

Resolution proofs are used in many practical applications for proof checking, debugging, core extraction, abstraction refinement, and interpolation. This paper presents and evaluates a practical method to obtain extended resolution proofs for conjoining BDDs in SAT solving. Our results enable the use of BDDs for these purposes instead—or in combination with—already established methods based on DPLL with clause learning.

As future work the ideas presented in this paper need to be extended to other BDD operations besides conjunction, particularly to existential quantification. We also conjecture that equivalence reasoning and, more generally, Gaussian elimination over GF(2), can easily be handled in the same way.

Finally we want to thank Eugene Goldberg for very fruitful discussions about the connection between extended resolution and BDDs.

References

1. M. Davis and H. Putnam. A computing procedure for quantification theory. *JACM*, 7, 1960.
2. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7), 1962.
3. J. P. Marques-Silva and K. A. Sakallah. GRASP — a new search algorithm for satisfiability. In *Proc. ICCAD'96*.
4. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. DAC'01*.
5. E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Proc. DATE'02*.
6. N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. SAT'03*.
7. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS'99*.
8. M. Velev and R. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. *J. Symb. Comput.*, 35(2), 2003.
9. I. Shlyakhter, R. Seater, D. Jackson, M. Sridharan, and M. Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. In *Proc. ASE'03*.
10. C. Sinz, A. Kaiser, and W. Küchlin. Formal methods for the validation of automotive product configuration data. *AI EDAM*, 17(1), 2003.
11. K. McMillan and N. Amla. Automatic abstraction without counterexamples. In *Proc. TACAS'03*.
12. Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Proc. POPL'05*.
13. K. McMillan. Interpolation and SAT-based model checking. In *Proc. CAV'03*, volume 2725 of *LNCS*.
14. L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proc. DATE'03*.
15. D. Motter and I. Markov. A compressed breath-first search for satisfiability. In *ALENEX'02*.
16. J. Franco, M. Kouril, J. Schlipf, J. Ward, S. Weaver, M. Dransfield, and W. Fleet. SBSAT: a state–based, BDD–based satisfiability solver. In *Proc. SAT'03*.
17. R. Damiano and J. Kukula. Checking satisfiability of a conjunction of BDDs. In *DAC'03*.
18. J. Huang and A. Darwiche. Toward good elimination orders for symbolic SAT solving. In *Proc. ICTAI'04*.
19. G. Pan and M. Vardi. Search vs. symbolic techniques in satisfiability solving. In *SAT'04*.
20. H.-S. Jin, M. Awedh, and F. Somenzi. CirCUs: A hybrid satisfiability solver. In *Proc. SAT'04*.
21. R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8), 1986.
22. T. E. Uribe and M. E. Stickel. Ordered binary decision diagrams and the Davis-Putnam procedure. In *Proc. Intl. Conf. on Constr. in Comp. Logics*, volume 845 of *LNCS*, 1994.
23. J. F. Groote and H. Zantema. Resolution and binary decision diagrams cannot simulate each other polynomially. *Discrete Applied Mathematics*, 130(2), 2003.
24. A. Urquhart. The complexity of propositional proofs. *Bulletin of the EATCS*, 64, 1998.
25. J. A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12, 1965.
26. G. Tseitin. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic*, 1970.

27. S. Cook. A short proof of the pigeon hole principle using extended resolution. *SIGACT News*, 8(4), 1976.
28. A. Haken. The intractability of resolution. *Theoretical Comp. Science*, 39, 1985.
29. P. Beame, H Kautz, and A Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res. (JAIR)*, 22, 2004.
30. F. Aloul, A. Ramani, I. Markov, and K. Sakallah. Solving difficult instances of boolean satisfiability in the presence of symmetry. *IEEE Trans. on Comp. Aided Design*, 22(9), 2003.
31. D. Plaisted, A. Biere, and Y. Zhu. A satisfiability procedure for quantified boolean formulae. *Discrete Applied Mathematics*, 130(2), 2003.
32. N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. SAT'05*, volume 3569 of *LNCS*.
33. F. Aloul, I. Markov, and K. Sakallah. FORCE: A fast and easy–to–implement variable-ordering heuristic. In *ACM Great Lakes Symp. on VLSI*, 2003.

A Example Extended Resolution Proof

To further illustrate proof generation and avoidance of tautological clauses, we give an example showing construction of part (ER-2a) of the ER proof. Let $S = (C, D)$ be an ordered clause set with $C = (ab)$ and $D = (\bar{a}b)$. The defining equations for the BDDs for C and D then are (using variable ordering $b > a$)

$$\begin{aligned} c &\leftrightarrow (b \mathbin{?} n_1 : c_0) & \qquad d &\leftrightarrow (b \mathbin{?} n_1 : d_0) \\ c_0 &\leftrightarrow (a \mathbin{?} n_1 : n_0) & \qquad d_0 &\leftrightarrow (a \mathbin{?} n_0 : n_1) \end{aligned}$$

and the resulting BDD h_2 denoting the conjunction of C and D is given by $h_2 \leftrightarrow (b \mathbin{?} n_1 : n_0)$. So the first part of our ER proof consists of introducing five new variables c, c_0, d, d_0 and h_2 together with their defining clauses by five applications of the extension rule. Then the resolution part follows. The last recursive step to derive $c \wedge d \rightarrow h_2$ looks like this:

$$\dfrac{\dfrac{(h_2 b \bar{n}_0) \qquad \dfrac{(\bar{d} b d_0) \qquad \dfrac{(\bar{c} b c_0) \qquad \dfrac{\vdots}{(\bar{c}_0 \bar{d}_0 n_0)}}{(\bar{c} b \bar{d}_0 n_0)}}{(\bar{c} \bar{d} b n_0)}}{(\bar{c} \bar{d} h_2 b)} \qquad \dfrac{\dfrac{\dfrac{(\bar{n}_1 \bar{n}_1 n_1) \qquad (\bar{c} \bar{b} n_1)}{(\bar{c} \bar{b} \bar{n}_1 n_1)} \qquad (\bar{d} \bar{b} n_1)}{(\bar{c} \bar{d} \bar{b} n_1)} \qquad (h_2 \bar{b} \bar{n}_1)}{(\bar{c} \bar{d} h_2 \bar{b})}}{(\bar{c} \bar{d} h_2)}$$

When we apply the rules stated above to avoid tautological clauses (three times rule (R1)), the whole proof for $c \wedge d \rightarrow h_2$ reduces to the following:

$$\dfrac{\dfrac{(h_2 b \bar{n}_0) \qquad \dfrac{(\bar{d} b d_0) \qquad \dfrac{(\bar{c} b c_0) \qquad \dfrac{(\bar{c}_0 a n_0) \qquad (\bar{d}_0 \bar{a} n_0)}{(\bar{c}_0 \bar{d}_0 n_0)}}{(\bar{c} b \bar{d}_0 n_0)}}{(\bar{c} \bar{d} b n_0)}}{(\bar{c} \bar{d} h_2 b)} \qquad \dfrac{(\bar{c} \bar{b} n_1) \qquad (h_2 \bar{b} \bar{n}_1)}{(\bar{c} h_2 \bar{b})}}{(\bar{c} \bar{d} h_2)}$$