Generating Extended Resolution Proofs with a BDD-Based SAT Solver

RANDAL E. BRYANT and MARIJN J. H. HEULE, Carnegie Mellon University, USA

In 2006, Biere, Jussila, and Sinz made the key observation that the underlying logic behind algorithms for constructing Reduced, Ordered Binary Decision Diagrams (BDDs) can be encoded as steps in a proof in the *extended resolution* logical framework. Through this, a BDD-based Boolean satisfiability (SAT) solver can generate a checkable proof of unsatisfiability. Such a proof indicates that the formula is truly unsatisfiable without requiring the user to trust the BDD package or the SAT solver built on top of it.

We extend their work to enable arbitrary existential quantification of the formula variables, a critical capability for BDD-based SAT solvers. We demonstrate the utility of this approach by applying a BDD-based solver, implemented by extending an existing BDD package, to several challenging Boolean satisfiability problems. Our results demonstrate scaling for parity formulas as well as the Urquhart, mutilated chessboard, and pigeonhole problems far beyond that of other proof-generating SAT solvers.

CCS Concepts: • **Theory of computation → Automated reasoning;**

Additional Key Words and Phrases: Boolean satisfiability, binary decision diagrams, extended resolution, Urquhart formulas, mutilated chessboard, pigeonhole problem

ACM Reference format:
Randal E. Bryant and Marijn J. H. Heule. 2023. Generating Extended Resolution Proofs with a BDD-Based SAT Solver. *ACM Trans. Comput. Logic* 24, 4, Article 31 (July 2023), 28 pages.
https://doi.org/10.1145/3595295

1 INTRODUCTION

When a Boolean satisfiability (SAT) solver returns a purported solution to a Boolean formula, its validity can easily be checked by making sure that the solution indeed satisfies the formula. When the formula is unsatisfiable, on the other hand, having the solver simply declare this to be the case requires the user to have faith in the solver, a complex piece of software that could well be flawed. Indeed, modern solvers employ a number of sophisticated techniques to reduce the search space. If one of those techniques is invalid or incorrectly implemented, the solver may overlook actual solutions and label a formula as unsatisfiable, even when it is not.

With SAT solvers providing the foundation for a number of different real-world tasks, this "false-negative" outcome could have unacceptable consequences. For example, when used as part of a formal verification system, the usual strategy is to encode some undesired property of the system as a formula. The SAT solver is then used to determine whether some operation of the system could

This work was supported by the U.S. National Science Foundation under grant number CCF-2108521.
Authors' address: R. E. Bryant and M. J. H. Heule, Carnegie Mellon University, Computer Science Department, Pittsburgh, Pennsylvania, USA, 15213; emails: {Randy.Bryant, mheule}@cs.cmu.edu.

1529-3785/2023/07-ART31 $15.00
https://doi.org/10.1145/3595295

lead to this undesirable property. Having the solver declare the formula to be unsatisfiable is an indication that the undesirable behavior cannot occur, but only if the formula is truly unsatisfiable.

Rather than requiring users to place their trust in a complex software system, a *proof-generating* solver constructs a proof that the formula is unsatisfiable. The proof has a form that can readily be checked by a simple proof checker. Initial work of checking unsatisfiability results was based on resolution proofs, but modern checkers are based on stronger proof systems [28, 52]. The checker provides an independent validation that the formula is indeed unsatisfiable. The checker can even be simple enough to be formally verified [19, 33, 45]. Such a capability has become an essential feature for modern SAT solvers.

In their 2006 papers [31, 44], Jussila, Sinz and Biere made the key observation that the underlying logic behind algorithms for constructing Reduced, Ordered Binary Decision Diagrams (BDDs) [6] can be encoded as steps in a proof in the *extended resolution* (ER) logical framework [46]. Through this, a BDD-based Boolean satisfiability solver can generate checkable proofs of unsatisfiability. Such proofs indicate that the formula is truly unsatisfiable without requiring the user to trust the BDD package or the SAT solver built on top of it.

In this article, we refine these ideas to enable a full-featured, BDD-based SAT solver. Chief among these is the ability to perform existential quantification on arbitrary variables. (Jussila, Sinz, and Biere [31] extended their original work [44] to allow existential quantification, but only for the root variable of a BDD.) In addition, we allow greater flexibility in the choice of variable ordering and the order in which conjunction and quantification operations are performed. This combination allows a wide range of strategies for creating a sequence of BDD operations that, starting with a set of input clauses, yield the BDD representation of the constant function **0**, indicating that the formula is unsatisfiable. Using the extended-resolution proof framework, these operations can generate a proof showing that the original set of clauses logically implies the empty clause, providing a checkable proof that the formula is unsatisfiable.

We evaluated the performance of both our SAT solver TBSAT and KISSAT, a state-of-the-art solver based on conflict detection and clause learning (CDCL) [5, 36]. Our results demonstrate that a proof-generating BDD-based SAT solver has very different performance characteristics from the more mainstream CDCL solvers. It does not do especially well as a general-purpose solver, but it can achieve far better scaling for several classic challenge problems [1, 17, 27, 48]. We find that several of these problems can be efficiently solved using the *bucket elimination* strategy [22] employed by Jussila, Sinz, and Biere [31], but others require a novel approach inspired by symbolic model checking [15].

This article assumes that the reader has some background in BDDs and their algorithms. This background can be obtained from a variety of tutorial presentations [2, 7, 8]. The article is largely self-contained regarding proof generation and is structured as follows. First, in Section 2, it provides a brief introduction to the resolution and extended resolution logical frameworks and to BDDs. Then, in Section 3, we show how a BDD-based SAT solver can generate proofs by augmenting algorithms for computing the conjunction of two functions represented as BDDs and for checking that one function logically implies another. We then describe our implementation and evaluate its performance on several classic problems in Section 4. We conclude with some general observations and suggestions for further work.

This article is an extended version of an earlier conference paper [12]. Here, we present more background material and more details about the proof-generation algorithms as well as updated benchmark results with a new implementation and on additional challenge problems.

2 PRELIMINARIES

Given a Boolean formula over a set of variables $\{x_1, x_2, \ldots, x_n\}$, a SAT solver attempts to find an assignment to these variables that will satisfy the formula or it declares the formula to be

unsatisfiable. As is standard practice, A *literal* ℓ can be either a variable or its complement. Most SAT solvers use Boolean formulas expressed in *conjunctive normal form*, where the formula consists of a set of *clauses*, each consisting of a set of literals. Each clause is a disjunction: if an assignment sets any of its literals to true, the clause is considered to be satisfied. The overall formula is a conjunction: a satisfying assignment must satisfy all of the clauses.

We write $\top$ to denote both tautology and logical truth. It arises when a clause contains both a variable and its complement. We write $\bot$ to denote logical falsehood. It is represented by an empty clause.

We make use of the *if-then-else* operation, written *ITE*, defined as $ITE(u, v, w) = (u \wedge v) \vee (\neg u \wedge w)$.

When writing clauses, we omit disjunction symbols and use overlines to denote negation, writing $\neg u \vee v \vee \neg w$ as $\overline{u}\, v\, \overline{w}$.

2.1 Resolution Proofs

Robinson [41] observed that the resolution inference rule formulated by Davis and Putnam [21] could form the basis for a refutation theorem-proving technique for first-order logic. Here, we consider its specialization to propositional logic. For clauses of the form $C \vee x$, and $\overline{x} \vee D$, the resolution rule derives the new clause $C \vee D$. This inference is written with a notation showing the required conditions above a horizontal line, and the resulting inference (known as the *resolvent*) below:

$$\frac{C \vee x \qquad \overline{x} \vee D}{C \vee D}.$$

Intuitively, the resolution rule is based on the property that implication is transitive. To see this, let proposition p denote $\neg C$ and proposition q denote D. Then, $C \vee x$ is equivalent to $p \rightarrow x$, $\overline{x} \vee D$ is equivalent to $x \rightarrow q$, and $C \vee D$ is equivalent to $p \rightarrow q$. In other words, the resolution rule encodes the property that if $p \rightarrow x$ and $x \rightarrow q$, then $p \rightarrow q$. As a special case, when C contains a literal ℓ and D contains its complement $\overline{\ell}$, then the resolvent of $C \vee x$ and $D \vee \overline{x}$ will be a tautology.

Resolution provides a mechanism for proving that a set of clauses is unsatisfiable. Suppose the input consists of m clauses. A resolution proof is given as a *trace* consisting of a series of *steps* S, where each step s_i consists of a clause C_i and a (possibly empty) list of antecedents A_i, where each antecedent is the index of one of the previous steps. The first set of steps, denoted S_m, consists of the input clauses without any antecedents. Each successive step then consists of a clause and a set of antecedents such that the clause can be derived from the clauses in the antecedents by one or more resolution steps. It follows by transitivity that for each step s_i, with $i > m$, clause C_i is logically implied by the input clauses, written $S_m \vDash C_i$. If, through a series of steps, we can reach a step s_t where C_t is the empty clause, then the trace provides a proof that $S_m \vDash \bot$, that is, the set of input clauses is not satisfiable.

A typical resolution proof contains many applications of the resolution rule. These enable deriving sequences of implications that combine by transitivity. For example, consider the following implications, shown both as formulas and as clauses:

Formula	Clause
$a \rightarrow b$	$\overline{a}\, b$
$x \rightarrow (b \rightarrow c)$	$\overline{x}\, \overline{b}\, c$
$c \rightarrow d$	$\overline{c}\, d$
$x \rightarrow (a \rightarrow d)$	$\overline{x}\, \overline{a}\, d$

We can derive the final clause from the first three using two resolution steps:

$$\dfrac{\dfrac{\overline{x}\,\overline{b}\,c \qquad \overline{a}\,b}{\overline{x}\,\overline{a}\,c} \qquad \overline{c}\,d}{\overline{x}\,\overline{a}\,d}$$

2.2 Reverse Unit Propagation (RUP)

Reverse unit propagation (RUP) provides an easily checkable way to express a linear sequence of resolution operations as a single proof step [25, 49]. It is the core rule supported by standard proof checkers [29, 51] for propositional logic. Let $C = \ell_1\,\ell_2\,\cdots\,\ell_p$ be a clause to be proved and let $D_1, D_2, \ldots, D_k$ be a sequence of supporting *antecedent* clauses occurring earlier in the proof. A RUP step proves that $\bigwedge_{1\le i\le k} D_i \rightarrow C$ by showing that the combination of the antecedents plus the negation of C leads to a contradiction. The negation of C is the formula $\overline{\ell}_1 \wedge \overline{\ell}_2 \wedge \cdots \wedge \overline{\ell}_p$ having a CNF representation consisting of p unit clauses of the form $\overline{\ell}_i$ for $1 \le i \le p$. A RUP check processes the clauses of the antecedent in sequence, inferring additional unit clauses. In processing clause D_i, if all but one of the literals in the clause is the negation of one of the accumulated unit clauses, then we can add this literal to the accumulated set. That is, all but this literal have been falsified, and so it must be set to true for the clause to be satisfied. The final step with clause D_k must cause a contradiction, that is, all of its literals are falsified by the accumulated unit clauses.

As an example, consider a RUP step to derive $x \rightarrow (a \rightarrow d)$ from the three clauses shown in the earlier example. A RUP proof would take the following form. Here, the target and antecedent clauses are listed along the top, while the resulting unit clauses are shown on the bottom, along with the final contradiction.

	Target	Antecedents		
Clause	$\overline{x}\,\overline{a}\,d$	$\overline{c}\,d$	$\overline{x}\,\overline{b}\,c$	$\overline{a}\,b$
Units	$x, a, \overline{d}$	$\overline{c}$	$\overline{b}$	$\bot$

RUP is an alternative formulation of resolution. For target clause C, it can be seen that applying resolution operations to the antecedent clauses from right to left will derive a clause C' such that $C' \subseteq C$. By *subsumption* [40], we then have $C' \rightarrow C$. Compared with listing each resolution operation as a separate step, using RUP as the basic proof step makes the proofs more compact.

2.3 Extended Resolution

Grigori S. Tseitin [46] introduced the extended-resolution proof framework in a presentation at the Leningrad Seminar on Mathematical Logic in 1966. The key idea is to allow the addition of new *extension* variables to a resolution proof in a manner that preserves the soundness of the proof. In particular, in introducing variable e, there must be an accompanying set of clauses that encode $e \leftrightarrow F$, where F is a formula over variables (both original and extension) that were introduced earlier [32]. These are referred to as the *defining clauses* for extension variable e. Variable e then provides a shorthand notation by which F can be referenced multiple times. Doing so can reduce the size of a clausal representation of a problem by an exponential factor.

An extension variable e is introduced into the proof by including its defining clauses in the list of clauses being generated. The proof checker must then ensure that the defining clauses obey the requirements for extension variables, as is discussed below. Thereafter, other clauses can include the extension variable or its complement, and they can list the defining clauses as antecedents.

Tseitin transformations are commonly used to encode a logic circuit or formula as a set of clauses without requiring the formulas to be "flattened" into a conjunctive normal form over the circuit inputs or formula variables. These introduced variables are called *Tseitin variables* and

are considered to be part of the input formula. An extended resolution proof takes this concept one step further by introducing additional variables as part of the proof. The proof checker must ensure that the extension variables are used in a way that does not result in an unsound proof. Some problems for which the minimum resolution proof must be of exponential size can be expressed with polynomial-sized proofs in extended resolution [18].

2.4 Clausal Proofs

We use a *clausal proof system* to validate our proofs based on the DRAT proof framework [29]. This framework supports both extended resolution and resolution operations based on a proof rule that generalizes reverse unit propagation. There are a number of fast and formally- verified checkers for these proofs [20, 33, 51]. The checker ensures that all extension variables are used properly and that each new clause can be derived via RUP from its antecedent clauses.

As in a resolution proof, a clausal proof is given as a trace, where each step s_i consists of a clause C_i and a list of antecedents A_i, where the initial m clauses are the input clauses. Let S_m denote the set of input clauses, and for $i > m$, define S_i inductively as $S_i = S_{i-1} \cup \{C_i\}$. The proof steps $s_{m+1}, \ldots, s_t$ represent a derivation from S_m to S_t. A clausal proof is a refutation if S_t contains the empty clause. Step s_i in a proof is valid if the equisatisfiability[1] of S_{i-1} and S_i can be checked using a polynomially decidable redundancy property. For the case in which C_i was obtained via RUP, we can simply perform a RUP check using C_i and the antecedents. In the case in which C_i is one of the defining clauses for some extension variable e, the checker must ensure that the clause is *blocked* [32]. That is, all possible resolvents of C_i with clauses in S_{i-1} that contain e must be tautologies. The blocked clause proof system is a generalization of extended resolution and allows the addition of blocked clauses that are blocked on non-extension variables. However, we do not use such capabilities in our proofs.

Clausal proofs also allow the removal of clauses. A proof can indicate that clause C_j can be removed after step s_i if it will not be used as an antecedent in any step s_k with $k > i$. With this restriction, clause deletion does not affect the integrity of the proof. As the experimental results of Section 5 demonstrate, deleting clauses that are no longer needed can substantially reduce the number of clauses the checker must track while processing a proof.

2.5 Binary Decision Diagrams

Reduced, Ordered Binary Decision Diagrams (which we refer to as simply "BDDs") provide a canonical form for representing Boolean functions and an associated set of algorithms for constructing them and testing their properties [6]. With BDDs, functions are defined over a set of variables $X = \{x_1, x_2, \ldots, x_n\}$. We let $\boldsymbol{T}_0$ and $\boldsymbol{T}_1$ denote the two leaf nodes, representing the constant functions $\mathbf{0}$ and $\mathbf{1}$, respectively.

Each nonterminal node $\boldsymbol{u}$ has an associated variable $\mathsf{Var}(\boldsymbol{u})$ and children $\mathsf{Hi}(\boldsymbol{u})$, indicating the case in which the node variable has value 1, and $\mathsf{Lo}(\boldsymbol{u})$, indicating the case in which the node variable has value 0.

Two lookup tables—the *unique table* and the *operation cache*—are critical for guaranteeing the canonicity of the BDDs and for ensuring polynomial performance of the BDD construction algorithms.

A node $\boldsymbol{u}$ is stored in a unique table, indexed by a key of the form $\langle \mathsf{Var}(\boldsymbol{u}), \mathsf{Hi}(\boldsymbol{u}), \mathsf{Lo}(\boldsymbol{u}) \rangle$, so that isomorphic nodes are never created. The nodes are shared across all of the BDDs [38]. In presenting algorithms, we assume a function GetNode$(x, \boldsymbol{u}_1, \boldsymbol{u}_0)$ that checks the unique table and either returns the node stored there or it creates a new node and enters it into the table. With this

[1]Two Boolean formulas are *equisatisfiable* if they are either both satisfiable or both unsatisfiable.

```
Apply(Op, u_1, ..., u_k)
    if IsTerminal(Op, u_1, ..., u_k):
        return TerminalValue(Op, u_1, ..., u_k)
    K ⟵ ⟨Op, u_1, ..., u_k⟩
    if K ∈ Cache:
        return Cache[K]
    w ⟵ ApplyRecur(Op, u_1, ..., u_k)
    Cache[K] ⟵ w
    return w
```

Fig. 1. General structure of the Apply algorithm. The operation for a specific logical operation Op is determined by its terminal cases and its recursive structure.

table, we can guarantee that the subgraphs with root nodes $\boldsymbol{u}$ and $\boldsymbol{v}$ represent the same Boolean function if and only if $\boldsymbol{u} = \boldsymbol{v}$. We can therefore uniquely identify Boolean functions with their BDD root nodes.

BDD packages support multiple operations for constructing and testing the properties of Boolean functions represented by BDDs. A number of these are based on the *Apply* algorithm [6]. Given a set of BDD roots $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_k$ representing functions $f_1, f_2, \ldots, f_k$, respectively, and a Boolean operation Op, the algorithm generates the BDD representation $\boldsymbol{w}$ of the operation applied to those functions. For example, with $k = 2$, and Op = And, Apply(And, $\boldsymbol{u}_1, \boldsymbol{u}_2$) returns the root node for the BDD representation of $f_1 \wedge f_2$.

Figure 1 shows pseudo-code describing the overall structure of the Apply algorithm. The details for a specific operation are embodied in the functions IsTerminal, TerminalValue, and ApplyRecur. The first two of these detect terminal cases and what value to return when a terminal case is encountered. The third describes how to handle the general case, in which the arguments must be expanded recursively. The algorithm makes use of *memoizing*, where previously computed results are stored in an operation cache, indexed by a key consisting of the operands [37]. Whenever possible, results are retrieved from this cache, avoiding the need to perform redundant calls to ApplyRecur. With this cache, the worst-case number of recursive steps required by the algorithm is bounded by the product of the sizes (in nodes) of the arguments.

3 PROOF GENERATION DURING BDD CONSTRUCTION

In our formulation, every newly created BDD node $\boldsymbol{u}$ is assigned an extension variable u. (Nodes are denoted by boldface characters, possibly with subscripts, e.g., $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{v}_1$, while their corresponding extension variables are denoted with a normal face, e.g., u, v, and v_1.) We then extend the Apply algorithm to generate proofs based on the recursive structure of the BDD operations.

Let S_m denote the set of input clauses. Our goal is to generate a proof that $S_m \vDash \bot$, that is, there is no satisfying assignment for these clauses. Our BDD-based approach generates a sequence of BDDs with root nodes $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_t$, where $\boldsymbol{u}_t = \boldsymbol{T}_0$, based on a combination of the following operations. (The exact sequencing of operations is determined by the *evaluation mechanism*, as is described in Section 5.)

(1) For input clause C_i, generate its BDD representation $\boldsymbol{u}_i$ using a series of Apply operations to perform the disjunctions.
(2) For roots $\boldsymbol{u}_j$ and $\boldsymbol{u}_k$, generate the BDD representation of their conjunction $\boldsymbol{u}_l = \boldsymbol{u}_j \wedge \boldsymbol{u}_k$ using the Apply operation to perform conjunction.
(3) For root $\boldsymbol{u}_j$ and some set of variables $Y \subseteq X$, perform existential quantification: $\boldsymbol{u}_k = \exists Y\, \boldsymbol{u}_j$.

Although the existential quantification operation is not mandatory for a BDD-based SAT solver, it can greatly improve its performance [23]. It is the BDD counterpart to Davis-Putnam variable

elimination on clauses [21]. As the notation indicates, there are often multiple variables that can be eliminated simultaneously. Although the operation can cause a BDD to increase in size, it generally causes a reduction. Our experimental results demonstrate the importance of this operation.

As these operations proceed, we simultaneously generate a set of proof steps. The details of each step are given later in the presentation. For each BDD generated, we maintain the proof invariant that the extension variable u_j associated with root node $\boldsymbol{u}_j$ satisfies $S_m \vDash u_j$.

(1) Following the generation of the BDD $\boldsymbol{u}_i$ for input clause C_i, we also generate a proof that $C_i \vDash u_i$. This is described in Section 3.1.
(2) Justifying the results of conjunctions requires two parts:
   (a) Using a modified version of the Apply algorithm for conjunction, we follow the structure of its recursive calls to generate a proof that the algorithm preserves implication: $u_j \wedge u_k \rightarrow u_l$. This is described in Section 3.2.
   (b) This implication can be combined with the earlier proofs that $S_m \vDash u_j$ and $S_m \vDash u_k$ to prove that $S_m \vDash u_l$.
(3) Justifying the quantification also requires two parts:
   (a) Following the generation of $\boldsymbol{u}_k$ via existential quantification, we perform a separate check that their associated extension variables satisfy $u_j \rightarrow u_k$. This check uses a proof-generating version of the Apply algorithm for implication checking. This is described in Section 3.3.
   (b) This implication can be combined with the earlier proof that $S_m \vDash u_j$ to prove that $S_m \vDash u_k$.

Compared with the prior work by Sinz and Biere [44], our key refinement is to handle arbitrary existential quantification operations. (When implementing a SAT solver, these quantifications must be applied in restricted ways [50], but since proofs of unsatisfiability only require proving implication, we need not be concerned with the details of these restrictions.) Rather than attempting to track the detailed logic underlying the quantification operation, we run a separate check that implication is preserved. As is the case with many BDD packages, our implementation can perform existential quantification of an arbitrary set of variables in a single pass over the argument BDD. We only need to perform a single implication check for the entire quantification.

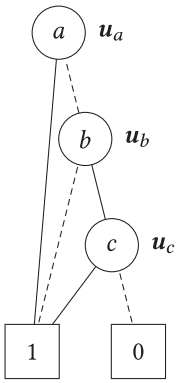
Sinz and Biere's construction assumed that there were special extension variables n_1 and n_0 to represent the BDD leaves $\boldsymbol{T}_1$ and $\boldsymbol{T}_0$. Their proofs then included unit clauses n_1 and $\overline{n}_0$ to force these variables to be set to true and false, respectively. We have found that these special variables are not required and instead directly associate leaves $\boldsymbol{T}_1$ and $\boldsymbol{T}_0$ with $\top$ and $\bot$, respectively.

The n variables in the input clauses all have associated BDD variables. The proof then introduces an extension variable u every time a new BDD node $\boldsymbol{u}$ is created. In the actual implementation, the extension variable (an integer) is stored as one of the fields in the node representation.

When creating a new node, the GetNode function adds (up to) four defining clauses for the associated extension variable. For node $\boldsymbol{u}$ with variable $\mathsf{Var}(\boldsymbol{u}) = x$, $\mathsf{Hi}(\boldsymbol{u}) = \boldsymbol{u}_1$, and $\mathsf{Lo}(\boldsymbol{u}) = \boldsymbol{u}_0$, the clauses are:

Notation	Formula	Clause
$\mathsf{HD}(\boldsymbol{u})$	$x \rightarrow (u \rightarrow u_1)$	$\overline{x}\,\overline{u}\,u_1$
$\mathsf{LD}(\boldsymbol{u})$	$\overline{x} \rightarrow (u \rightarrow u_0)$	$x\,\overline{u}\,u_0$
$\mathsf{HU}(\boldsymbol{u})$	$x \rightarrow (u_1 \rightarrow u)$	$\overline{x}\,\overline{u}_1\,u$
$\mathsf{LU}(\boldsymbol{u})$	$\overline{x} \rightarrow (u_0 \rightarrow u)$	$x\,\overline{u}_0\,u$

The names for these clauses combine an indication of whether they correspond to variable x being 1 (H) or 0 (L) and whether they form an implication from the node down to its child (D) or from



Clause	Target u_a	HU(u_a) $\overline{a}\,u_a$	LU(u_a) $a\,\overline{u}_b\,u_a$	LU(u_b) $b\,u_b$	HU(u_b) $\overline{b}\,\overline{u}_c\,u_b$	HU(u_c) $\overline{c}\,u_c$	C $a\,\overline{b}\,c$
Units	$\overline{u}_a$	$\overline{a}$	$\overline{u}_b$	b	$\overline{u}_c$	$\overline{c}$	$\bot$

Fig. 2. BDD representation of clause $C = a\,\overline{b}\,c$ and the justification of root unit clause $\boldsymbol{u}_a$ with one RUP step.

the child up to its parent (U). When one of the child nodes $\boldsymbol{u}_0$ or $\boldsymbol{u}_1$ is a leaf, some of these defining clauses will degenerate into tautologies and some will reduce to just two literals. Tautologies are not included in the proof. These defining clauses encode the assertion

$$u \quad \leftrightarrow \quad ITE(x, u_1, u_0),$$

satisfying Tseitin's restriction on the use of extension variables. Each clause is numbered according to its step number in the trace.

3.1 Generating BDD Representations of Clauses

The BDD representation for a clause C has a simple, linear structure. For root node $\boldsymbol{u}$, it is easy to prove that $C \vDash u$ using one RUP step. The general algorithm is described by Sinz and Biere [44]. Here, we illustrate the idea via an example.

Figure 2 shows the BDD representation of clause $C = a\,\overline{b}\,c$. As can be seen, the BDD for a clause has a very specific structure. For each literal in the clause, there is a node labeled by the variable, with one child being leaf $\boldsymbol{T}_1$ and the other being either the node for the next literal in the variable ordering or leaf $\boldsymbol{T}_0$. The lower part of the figure shows a RUP justification of $C \vDash u_a$, where $\boldsymbol{u}_a$ is the root node of the BDD. The proof uses the antecedents HU($\boldsymbol{u}$) and LU($\boldsymbol{u}$) for each node $\boldsymbol{u}$ in the BDD (except for the tautological case representing the final edge to $\boldsymbol{T}_0$), with the final antecedent being the clause itself. The RUP steps introduce the complements of the clause variables as unit clauses, causing a contradiction with the input clause. The order in which the two defining clauses for a node are listed in the antecedent depends on whether the variable is positive or negative in the clause. As this example demonstrates, we can generate a single proof step for $C_i \vDash u_i$ for each input clause C_i.

3.2 Performing Conjunctions

The key idea in generating proofs for the conjunction operation is to follow the recursive structure of the Apply algorithm. We do this by integrating proof generation into the Apply procedures, as is shown in Figure 3. This follows the standard form of the Apply algorithm (Figure 1), with the novel feature that each result includes both a BDD node $\boldsymbol{w}$ and a proof step number s. For arguments

Terminal Cases	
Condition	Result
$\boldsymbol{u} = \boldsymbol{v}$	$(\boldsymbol{u}, \top)$
$\boldsymbol{u} = \boldsymbol{T}_0$	$(\boldsymbol{T}_0, \top)$
$\boldsymbol{v} = \boldsymbol{T}_0$	$(\boldsymbol{T}_0, \top)$
$\boldsymbol{u} = \boldsymbol{T}_1$	$(\boldsymbol{v}, \top)$
$\boldsymbol{v} = \boldsymbol{T}_1$	$(\boldsymbol{u}, \top)$

```
ApplyRecur(And, u, v)
    J ⟵ {}
    x ⟵ min(Var(u), Var(v))
    if x = Var(u):
        u1, u0 ⟵ Hi(u), Lo(u)
        J ⟵ J ∪ {HD(u), LD(u)}
    else:  u1, u0 ⟵ u, u
    if x = Var(v):
        v1, v0 ⟵ Hi(v), Lo(v)
        J ⟵ J ∪ {HD(v), LD(v)}
    else:  v1, v0 ⟵ v, v
    w1, s1 ⟵ Apply(And, u1, v1)
    w0, s0 ⟵ Apply(And, u0, v0)
    J ⟵ J ∪ {s1, s0}
    if w1 = w0:
        w ⟵ w1
    else:
        w ⟵ GetNode(x, w1, w0)
        J ⟵ J ∪ {HU(w), LU(w)}
    s ⟵ JustifyAnd(⟨u, v, w⟩, J)
    return (w, s)
```

Fig. 3. Terminal cases and recursive step of the Apply operation for conjunction, modified for proof generation. Each call returns both a node and a proof step.

$\boldsymbol{u}$ and $\boldsymbol{v}$, step s lists clause $\overline{u}\,\overline{v}\,w$ along with antecedents defining a RUP proof of the implication $u \wedge v \rightarrow w$.

As the table of terminal cases shows, these cases all correspond to tautologies. For example, the case of $\boldsymbol{u} = \boldsymbol{T}_1$, giving $\boldsymbol{w} = \boldsymbol{v}$ is justified by that tautology $\top \wedge v \rightarrow v$. Failing a terminal or previously computed case, the function must recurse, branching on the variable x that is the minimum of the two root variables. The procedure accumulates a set of proof steps J to be used in the implication proof. These include the two steps (possibly tautologies) from the two recursive calls. At the end, it invokes a function JustifyAnd to generate the required proof. In returning the pair $(\boldsymbol{w}, s)$, this value will be stored in the operation cache and returned as the result of the Apply operation.

3.2.1 Proof Generation for the Standard Case. A proof generated by Apply with operation And inducts on the structure of the argument and result BDDs. That is, it assumes that the result nodes $\boldsymbol{w}_1$ and $\boldsymbol{w}_0$ of the recursive calls to arguments $\boldsymbol{u}_1$ and $\boldsymbol{v}_1$ and to $\boldsymbol{u}_0$ and $\boldsymbol{v}_0$ satisfy the implications $u_1 \wedge v_1 \rightarrow w_1$ and $u_0 \wedge v_0 \rightarrow w_0$, and that these calls generated proof steps s_1 and s_0 justifying these implications. For the standard case, in which none of the equalities hold and the recursive calls do not yield tautologies, the supporting clauses for the proof are shown in Figure 4. That is, the set J contains references to eight clauses, which we identify by labels. Six of these are defining clauses: the downward clauses for the argument nodes (labeled uhd, vhd, uld, and vld) and the upward clauses for the result (labeled whu and wlu). The other two are implications for the two recursive calls, labeled (andh and andl). We partition these supporting clauses into two sets:

$$A_H = \text{uhd, vhd, whu, andh} \quad (1)$$

$$A_L = \text{uld, vld, wlu, andl} \quad (2)$$

Label	Formula	Clause
UHD	$\mathsf{HD}(\boldsymbol{u})$	$\overline{x}\,\overline{u}\,u_1$
ULD	$\mathsf{LD}(\boldsymbol{u})$	$x\,\overline{u}\,u_0$
VHD	$\mathsf{HD}(\boldsymbol{v})$	$\overline{x}\,\overline{v}\,v_1$
VLD	$\mathsf{LD}(\boldsymbol{v})$	$x\,\overline{v}\,v_0$
WHU	$\mathsf{HU}(\boldsymbol{w})$	$\overline{x}\,\overline{w}_1\,w$
WLU	$\mathsf{LU}(\boldsymbol{w})$	$x\,\overline{w}_0\,w$
ANDH	$u_1 \wedge v_1 \rightarrow w_1$	$\overline{u}_1\,\overline{v}_1\,w_1$
ANDL	$u_0 \wedge v_0 \rightarrow w_0$	$\overline{u}_0\,\overline{v}_0\,w_0$

Fig. 4. Supporting clauses for standard step of the Apply algorithm for conjunction operations.

Clause	Target $\overline{x}\,\overline{u}\,\overline{v}\,w$		UHD $\overline{x}\,\overline{u}\,u_1$	VHD $\overline{x}\,\overline{v}\,v_1$	WHU $\overline{x}\,\overline{w}_1\,w$	ANDH $\overline{u}_1\,\overline{v}_1\,w_1$
Units	$x, u, v, \overline{w}$		u_1	v_1	$\overline{w}_1$	$\bot$
Clause	Target $\overline{u}\,\overline{v}\,w$	Previous $\overline{x}\,\overline{u}\,\overline{v}\,w$	ULD $x\,\overline{u}\,u_0$	VLD $x\,\overline{v}\,v_0$	WLU $x\,\overline{w}_0\,w$	ANDL $\overline{u}_0\,\overline{v}_0\,w_0$
Units	$u, v, \overline{w}$	$\overline{x}$	u_0	v_0	$\overline{w}_0$	$\bot$

Fig. 5. RUP proof steps for standard recursive step of the conjunction operation.

These supporting clauses are used to derive the target clause $u \wedge v \rightarrow w$ using the two RUP steps shown in Figure 5. The first step proves the weaker target $x \rightarrow (u \wedge v \rightarrow w)$, having clausal representation $\overline{x}\,\overline{u}\,\overline{v}\,w$ using the supporting clauses in A_H. The second step proves the full target, having clausal representation $\overline{u}\,\overline{v}\,w$. It uses both the weaker result and the supporting clauses in A_L.

3.2.2 Proof Generation for Special Cases. The proof structure shown in Figure 5 only holds for the standard form of the recursion. However, there are many special cases, such as when a recursive call yields a tautologous result, when some of the child nodes are equal, and when the two recursive calls return the same node. Fortunately, a general approach can handle the many special cases that arise. The examples shown in Figure 6 illustrate a range of possibilities. Based on these and the standard case of Figure 5, we show how to handle all of the cases with a simple algorithm.

Figure 6(A) illustrates the case in which some of the nodes in the recursive calls are equal. In particular, when $\mathsf{Var}(\boldsymbol{u}) > \mathsf{Var}(\boldsymbol{v})$, the recursion will split, with $\boldsymbol{u}_1 = \boldsymbol{u}_0 = \boldsymbol{u}$. This will cause supporting clauses UHD and ULD to be tautologies. This example also has $\boldsymbol{w}_1 = \boldsymbol{w}_0 = \boldsymbol{w}$, as will occur when the two recursive calls return the identical result. This will cause supporting clauses WHU and WLU to be tautologies. The two sets of equalities will cause supporting clause ANDH to be $\overline{u}\,\overline{v}_1\,\overline{w}$ and supporting clause ANDL to be $\overline{u}\,\overline{v}_0\,\overline{w}$. As can be seen, the resulting proof will consist of the same two steps as the standard form, but with fewer supporting clauses.

Figure 6(B) illustrates the case in which $\boldsymbol{u}_1 = \boldsymbol{T}_1$ and, therefore, the first recursive call generates a tautologous result. This case will cause $\boldsymbol{w}_1 = \boldsymbol{v}_1$ and, therefore, supporting clause WHU will be $\overline{x}\,\overline{v}_1\,w$. In addition, supporting clauses UHD and ANDH will be tautologies. Despite these changes, the proof will still have the same two-step structure as the standard case.

Finally, Figure 6(C) illustrates the case in which $\boldsymbol{u}_1 = \boldsymbol{T}_0$ and, therefore, the first recursive call again generates a tautologous result. This case will cause $\boldsymbol{w}_1 = \boldsymbol{T}_0$, and only two clauses among

(A) $\boldsymbol{u}_1 = \boldsymbol{u}_0$ and $\boldsymbol{w}_1 = \boldsymbol{w}_0$

Clause	Target $\overline{x}\,\overline{u}\,\overline{v}\,w$	VHD $\overline{x}\,\overline{v}\,v_1$	ANDH $\overline{u}\,\overline{v}_1\,w$
Units	$x, u, v, \overline{w}$	v_1	$\bot$

Clause	Target $\overline{u}\,\overline{v}\,w$	Previous $\overline{x}\,\overline{u}\,\overline{v}\,w$	VLD $x\,\overline{v}\,v_0$	ANDL $\overline{u}\,\overline{v}_0\,w$
Units	$u, v, \overline{w}$	$\overline{x}$	v_0	$\bot$

(B) $\boldsymbol{u}_1 = \boldsymbol{T}_1$

Clause	Target $\overline{x}\,\overline{u}\,\overline{v}\,w$	VHD $\overline{x}\,\overline{v}\,v_1$	WHU $\overline{x}\,\overline{v}_1\,w$
Units	$x, u, v, \overline{w}$	v_1	$\bot$

Clause	Target $\overline{u}\,\overline{v}\,w$	Previous $\overline{xu}\,\overline{v}\,w$	ULD $x\,\overline{u}\,u_0$	VLD $x\,\overline{v}\,v_0$	WLU $x\,\overline{w}_0\,w$	ANDL $\overline{u}_0\,\overline{v}_0\,w_0$
Units	$u, v, \overline{w}$	$\overline{x}$	u_0	v_0	$\overline{w}_0$	$\bot$

(C) $\boldsymbol{u}_1 = \boldsymbol{T}_0$

Clause	Target $\overline{u}\,\overline{v}\,w$	UHD $\overline{x}\,\overline{u}$	ULD $x\,\overline{u}\,u_0$	VLD $x\,\overline{v}\,v_0$	WLU $x\,\overline{w}_0\,w$	ANDL $\overline{u}_0\,\overline{v}_0\,w_0$
Units	$u, v, \overline{w}$	$\overline{x}$	u_0	v_0	$\overline{w}_0$	$\bot$

Fig. 6. RUP proof steps for conjunction for illustrative special cases.

those in A_H will not be tautologies: UHD will be $\overline{x}\,\overline{u}$ and VHD will be $\overline{x}\,\overline{v}\,v_1$. As can be seen, the proof for this case consists of a single RUP step. Furthermore, it does not make use of supporting clause VHD.

These three examples illustrate the following general properties:

- When neither ANDH nor ANDL is a tautology, the proof requires two steps. Some of the supporting clauses may be tautologies, but the proof can follow the standard form shown in in Figure 5.
- When either ANDH or ANDL is a tautology, it may be possible to generate a single-step proof. Otherwise, it can follow the standard, two-step form.

Given these possibilities, our implementation of JustifyAnd uses the following strategy:

(1) If supporting clause ANDH is a tautology, then attempt a single-step proof, using the non-tautologous clauses in A_H followed by those in A_L. If this fails, then perform a two-step proof.
(2) Similarly, if supporting clause ANDL is a tautology, then attempt a single-step proof, using the non-tautologous clauses in A_L followed by those in A_H. If this fails, then perform a two-step proof.
(3) A two-step proof proceeds by first proving the weaker clause $\overline{x}\,\overline{u}\,\overline{v}\,w$ using the non-tautologous clauses in A_H. It then uses this result plus the clauses in A_L to justify target clause $\overline{u}\,\overline{v}\,w$.

In all cases, the antecedent is generated by stepping through the clauses in their specified order, adding only those that cause unit propagation or conflict.

Terminal Cases	
Condition	Result
$\boldsymbol{u} = \boldsymbol{v}$	$\top$
$\boldsymbol{u} = T_0$	$\top$
$\boldsymbol{v} = T_1$	$\top$
$\boldsymbol{u} = T_1, \boldsymbol{v} \neq T_1$	Error
$\boldsymbol{v} = T_0, \boldsymbol{u} \neq T_0$	Error

```
ApplyRecur(Imply, u, v)
        J ⟵ {}
        x ⟵ min(Var(u), Var(v))
        if x = Var(u):
                u1, u0 ⟵ Hi(u), Lo(u)
                J ⟵ J ∪ {HD(u), LD(u)}
        else:   u1, u0 ⟵ u, u
        if x = Var(v):
                v1, v0 ⟵ Hi(v), Lo(v)
                J ⟵ J ∪ {HU(v), LU(v)}
        else:   v1, v0 ⟵ v, v
        s1 ⟵ Apply(Imply, u1, v1)
        s0 ⟵ Apply(Imply, u0, v0)
        J ⟵ J ∪ {s1, s0}
        s ⟵ JustifyImplication(⟨u, v⟩, J)
        return s
```

Fig. 7. Terminal cases and recursive step of the Apply algorithm for implication checking.

Label	Formula	Clause
UHD	$\mathsf{HD}(\boldsymbol{u})$	$\overline{x}\,\overline{u}\,u_1$
ULD	$\mathsf{LD}(\boldsymbol{u})$	$x\,\overline{u}\,u_0$
VHU	$\mathsf{HU}(\boldsymbol{v})$	$\overline{x}\,\overline{v}_1\,v$
VLU	$\mathsf{LU}(\boldsymbol{v})$	$x\,\overline{v}_0\,v$
IMH	$u_1 \rightarrow v_1$	$\overline{u}_1\,v_1$
IML	$u_0 \rightarrow v_0$	$\overline{u}_0\,v_0$

Fig. 8. Clause structure for the standard step of implication checking.

3.3 Checking Implication

As described in Section 3, we need not track the detailed logic of the algorithm that performs existential quantification. Instead, when the quantification operation applied to node $\boldsymbol{u}$ generates node $\boldsymbol{v}$, we generate a proof of implication afterwards, using the Apply algorithm adapted for implication checking, as shown in Figure 7. A failure of this implication check would indicate an error in the BDD package. Thus, its only purpose is to generate a proof that the implication holds, signaling a fatal error if the implication does not hold.

This particular operation does not generate any new nodes; thus, the returned result is simply a proof step number. The (successful) terminal cases correspond to the tautological cases $u \rightarrow u$, $\bot \rightarrow v$, and $u \rightarrow \top$.

Each recursive step accumulates up to six proof steps as the set J to be used in the implication proof. Figure 8 shows the structure of these clauses for the standard case in which neither equality holds and neither recursive call returns $\top$. The clauses consist of the two downward defining clauses for argument $\boldsymbol{u}$, labeled UHD and ULD, the two upward defining clauses for argument $\boldsymbol{v}$, labeled VHU and VLU, and the clauses returned by the recursive calls, labeled IMH and IML.

Figure 9 shows the two RUP steps required to prove the standard case. The first step proves the weaker target $x \rightarrow (u \rightarrow v)$, having clausal representation $\overline{x}\,\overline{u}\,v$ using the three supporting clauses containing $\overline{x}$. The second proves the full target, having clausal representation $\overline{u}\,v$ using the weaker result plus the supporting clauses containing x.

Clause	Target $\overline{x}\,\overline{u}\,v$		UHD $\overline{x}\,\overline{u}\,u_1$	VHU $\overline{x}\,\overline{v}_1\,v$	IMH $\overline{u}_1\,v_1$
Units	$x, u, \overline{v}$		u_1	$\overline{v}_1$	$\bot$
Clause	Target $\overline{u}\,v$	Previous $\overline{x}\,\overline{u}\,v$	ULD $x\,\overline{u}\,u_0$	VLU $x\,\overline{v}_0\,v$	IML $\overline{u}_0\,v_0$
Units	$u, \overline{v}$	$\overline{x}$	u_0	$\overline{v}_0$	$\bot$

Fig. 9. RUP proof steps for standard recursive check of implication checking.

As with the conjunction operation, there can be many special cases, but they can be handled with the same general strategy. If either recursive result IMH or IML is a tautology, a one-step proof is attempted. If that fails, or if neither recursive result is a tautology, a two-step proof is generated.

4 IMPLEMENTATION

We implemented the TBUDDY proof-generating BDD package by modifying the widely used BUDDY BDD package, developed by Jørn Lind-Nielsen in the 1990s [9]. This involved adding several additional fields to the BDD node and cache entry data structures, yielding a total memory overhead of 1.35×. TBUDDY generates proofs in the LRAT proof format [19]. We then implemented TBSAT, a proof-generating SAT solver based on TBUDDY.

TBSAT supports three different evaluation mechanisms:

Linear: Forms the conjunction of the clauses. No quantification is performed. This mode matches the operation described for the original version of EBDDRES [44]. When forming the conjunction of a set of terms, the program makes use of a first-in, first-out queue, removing two elements from the front of the queue, computing their conjunction, and placing the result at the end of the queue. This has the effect of forming a binary tree of conjunctions.

Bucket Elimination: Place the BDDs representing the clauses into buckets according to the levels of their topmost variables. Then process the buckets from lowest to highest. While a bucket has more than one element, repeatedly remove two elements, form their conjunction, and place the result in the bucket designated by its topmost variable. Once the bucket has a single element, existentially quantify the topmost variable and place the result in the appropriate bucket [22]. This matches the operation described for the revised version of EBDDRES [31]. It provides a systematic way to perform an analog to the *early quantification* method of symbolic model checking [14].

Scheduled: Perform operations as specified by a *scheduling* file, as described below.

The scheduling file contains a sequence of lines, each providing a command in a simple, stack-based notation:

c $c_1, \ldots, c_k$	Push the BDD representations of the specified clauses onto the stack
a m	Replace the top m elements on the stack with their conjunction
q $v_1, \ldots, v_k$	Replace the top stack element with its quantification by the specified variables

5 EXPERIMENTAL RESULTS

In our preliminary experiments, we found that the capabilities of TBSAT differ greatly from the more mainstream CDCL solvers. Therefore, it must be evaluated by a different set of standards. In particular, CDCL solvers are most commonly evaluated according to their performance on collections of benchmark problems in a series of annual solver competitions. Over the years, the

benchmark problems have been updated to provide new challenges and to better distinguish the performance of the different solvers. This competition has stimulated major improvements in the solvers through improved algorithms and implementation techniques. One unintended consequence, however, has been that the benchmarks have evolved to be only at, or slightly beyond, the capabilities of CDCL solvers.

As an example, Chatalic and Simon [16] and Li [34] contributed multiple benchmark formulas for the 2002 SAT competition [42] based on a class of unsatisfiable formulas devised by Urquhart [48]. (These are described in Section 5.2.) The formulas scale quadratically by a size parameter m both in terms of the number of variables and the number of clauses. Simon's largest benchmark had $m = 5$, while Li's had $m = 4$. No solver at the time could complete for these formulas, even though Li's formula for $m = 4$ has only 288 variables and 768 clauses. The 2022 SAT competition featured a special "Anniversary track" using as formulas the 5355 formulas that have been used across all prior SAT competitions. In all, 32 solvers participated in the competition with a 5000-second time limit for each problem. Even after years of improvements in the solvers and with vastly better hardware, none of the solvers completed these 20-year-old benchmark problems. There has been no attempt to evaluate solvers running on Urquhart formulas for larger values of m because these were clearly beyond the reach of the competing solvers.

By contrast, TBSAT can easily handle the Urquhart formulas. Generating proofs of unsatisfiability for Simon's benchmark with $m = 5$ and Li's benchmark with $m = 4$ requires 0.23 and 0.13 seconds, respectively. We show experimental results with $m = 38$ for Li's version and $m = 60$ for Simon's. In a more recent effort [9], we augmented TBSAT to use Gaussian elimination for reasoning about parity constraints, allowing us to generate an unsatisfiability proof for Li's version with $m = 316$, a formula with over two million variables and five million clauses. This example demonstrates that measuring performance on benchmarks designed to evaluate CDCL solvers cannot capture the full capabilities of a BDD-based SAT solver.

In the following experiments, we explore the capability of TBSAT on four scalable benchmark problems that pose major challenges for CDCL solvers. These do not to show that BDD-based methods are uniformly superior to CDCL, but rather that they can perform very well on some classes of problems for which CDCL is especially weak. A long-term research direction is to combine the capabilities of CDCL and BDDs to build on the strengths of each.

All experiments were performed on a 3.2 GHz Apple M1 Max processor with 64 GB of memory and running the OS X operating system. The runtime for each experiment was limited to 1000 seconds. We compare the performance of TBSAT to that of KISSAT, the winner of several recent SAT solver competitions [5]. KISSAT represents the state-of-the-art in CDCL solvers. The proofs were checked using DRAT-TRIM for the proofs generated by KISSAT and LRAT-CHECK for those generated by TBSAT. We report both the elapsed time by the solver and the total number of clauses in the proof of unsatisfiability. For KISSAT, the proof clauses indicate the conflicts the solver encountered during its search. For TBSAT, these are the defining clauses for the extension variables (up to four per BDD node generated) and the derived clauses (one per input clause and up to two per result inserted into the operation cache.)

5.1 Reordered Parity Formulas

Chew and Heule [17] introduced a benchmark problem based on computing the parity of a set of Boolean values $x_1, \ldots, x_n$ using two different orderings of the inputs and with one of the variables negated in the second computation:

$$\begin{aligned} \mathit{ParityA}(x_1, \ldots, x_n) &= x_1 \oplus x_2 \oplus \cdots \oplus x_n \\ \mathit{ParityB}(x_1, \ldots, x_n) &= \left[p_1 \oplus x_{\pi(1)}\right] \oplus \left[p_2 \oplus x_{\pi(2)}\right] \oplus \cdots \oplus \left[p_n \oplus x_{\pi(n)}\right], \end{aligned}$$

where π is a random permutation, and each p_i is either 0 or 1, with the restriction that $p_i = 1$ for only one value of i. The two sums associate from left to right. Therefore, the formula *ParityA* $\wedge$ *ParityB* is unsatisfiable, but the permutation makes this difficult for CDCL solvers to determine. The CNF has a total of $3n - 2$ variables: n values of x_i, plus the auxiliary variables encoding the intermediate terms in the two expressions.

Chew and Heule experimented with the CDCL solver CaDiCaL [3] and found it could not handle cases with n greater than 50. They devised a specialized method for directly generating proofs in the DRAT proof system, obtaining proofs that scale as $O(n \log n)$, and gave results for up to $n = 4{,}000$. They also tried EBDDRES, but only in its default mode, where it performs only linear evaluation without any quantification.

Figure 10 shows the result of applying both TBSAT and KISSAT to this problem. In this and other figures, the top graph shows how the runtime scales with the problem size, whereas the bottom graph shows how the number of proof clauses scale. Both graphs are log-log plots; thus, the values are highly compressed along both dimensions. Linear evaluation performs poorly, only handling up to $n = 24$ within the 1000-second time limit, generating a proof with over 312 million clauses. Using KISSAT, we found that the results were very sensitive to the choice of random permutation. Thus, we show results using three different random seeds for each value of n. We were able to generate proofs for instances with n up to 46 within the time limit but also started having timeouts with $n = 42$. We can see that KISSAT does better than linear evaluation with TBSAT, but both appear to scale exponentially.

Bucket elimination, on the other hand, displays much better scaling. We found that the best performance was achieved by randomly permuting the variables, although this strategy only yields a constant factor improvement over the ordering from the CNF file. As the graphs show, we were able to handle cases with n up to 9,750, within the time limit. This generated a proof with over 419 million clauses, but the LRAT checker was able to verify this proof in 256 seconds. Although TBSAT could generate proofs for larger values of n, these exceeded the capacity of the LRAT checker.

Included in the second graph are results for running Chew and Heule's proof generator on this problem. As can be seen, the proof sizes generated by TBSAT are comparable to theirs up to around $n = 100$. From there on, however, the benefit of their $O(n \log n)$ algorithm becomes apparent. Even for $n = 10{,}000$, their proof contains less than 11 million clauses. Of course, their construction relies on particular properties of the underlying problem, while ours was generated by a general-purpose SAT solver.

5.2 Urquhart Formulas

Urquhart [48] introduced a family of formulas that require resolution proofs of exponential size. Over the years, two families of SAT benchmarks have been labeled as "Urquhart Problems": one developed by Chatalic and Simon [16], and the other by Li [34]. These are considered to be difficult challenge problems for SAT solvers. Here, we define their general form, describe the differences between the two families, and evaluate the performance of both KISSAT and TBSAT on both classes.

Urquhart's construction is based on a class of bipartite graphs with special properties. Define G_k as the set of undirected graphs, with each graph satisfying the following properties:

- It is *bipartite*: The set of vertices can be partitioned into sets L and R such that the edges E satisfy $E \subseteq L \times R$.
- It is *balanced*: $|L| = |R|$.
- It has bounded degree: No vertex has more than k incident edges.

Furthermore, the graphs must be *expanders*, defined as follows [30]. For a subset of vertices $U \subseteq L$, define $R(U)$ to be those vertices in R adjacent to the vertices on U. A graph in G_k is an expander

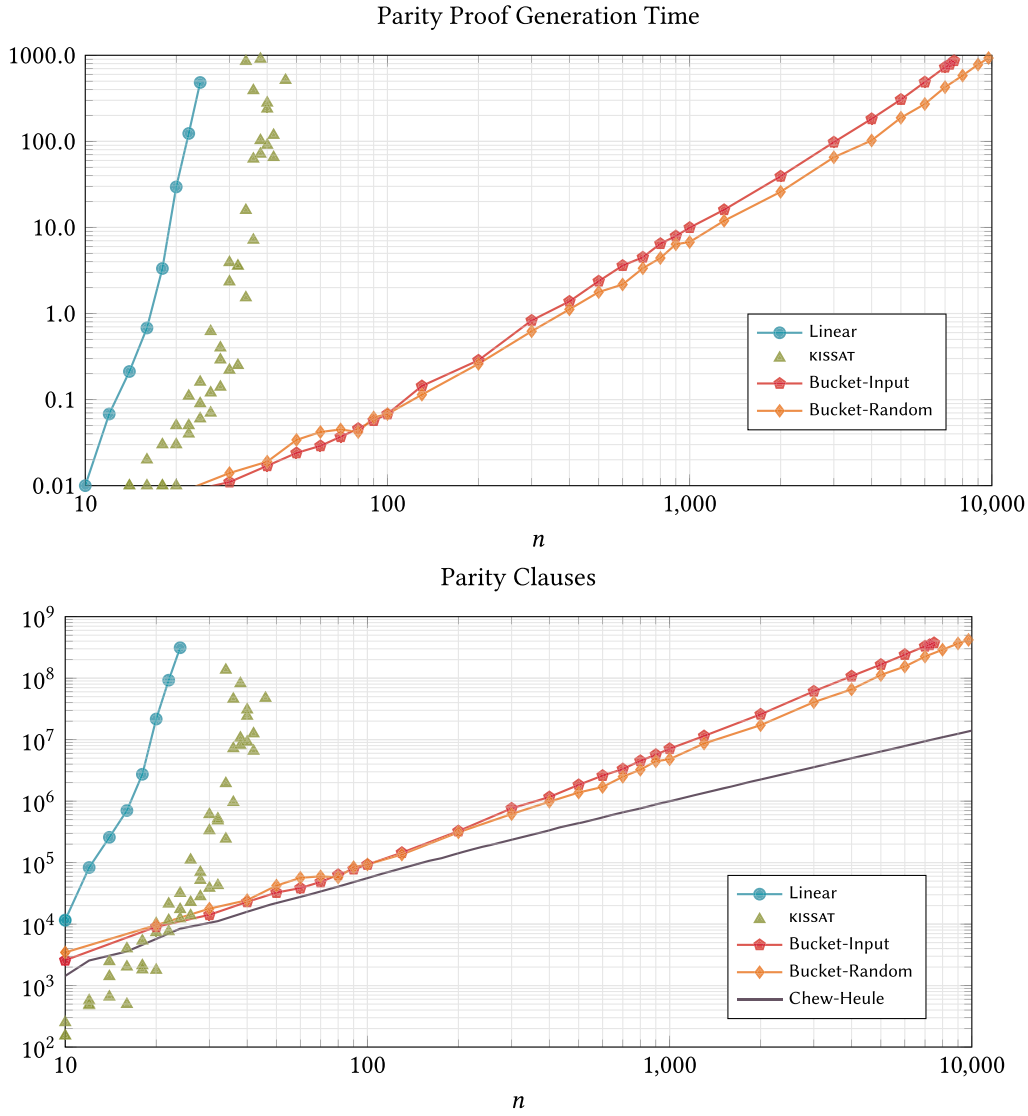


Fig. 10. Generating unsatisfiability proofs for reordered parity encodings with n data variables.

if there is some constant $d > 0$ such that for any $U \subset L$ with $|U| \leq |L|/2$, the set $R(U)$ satisfies $|R(U)| \geq (d+1)|U|$. Urquhart considers expander graphs with degree bound $k = 5$ and that are parameterized by a size value m with $|L| = |R| = m^2$.

To transform such a graph into a formula, each edge $(i, j) \in E$ has an associated variable $x_{\{i,j\}}$. (We use this notation to emphasize that the order of the indices does not matter.) Each vertex is assigned a polarity $p_i \in 0, 1$ such that the sum of the polarities is odd. The clauses then encode the formula:

$$\sum_{i=1}^{2m^2} \left[\sum_{(i,j)\in E} x_{\{i,j\}} + p_i \right] \equiv 0 \pmod 2.$$

This is false, of course, since each edge gets counted twice in the sum and the sum of the polarities is odd.

The two families of benchmarks differ in how the graphs are constructed. Li's benchmarks are based on the explicit construction of expander graphs due to Margulis [24, 35] that is cited by Urquhart. Thus, his graphs are fully defined by the size parameter m. Simon's benchmarks are based on randomly generated graphs. Thus, they are characterized by both the size parameter m and the initial random seed s. Although random graphs satisfy the expander condition with high probability [30], it is unlikely that the particular instances generated by Simon's benchmark generator are truly expander graphs. The widely used SAT benchmarks with names of the form `Urq`M`_`S`.cnf` were generated by Simon's program for size parameter $m = M$ and initial seed S. For Simon's benchmarks, we used five different seeds for each value of m.

Figure 11 shows data for running KISSAT as well as TBSAT using bucket elimination. The data for KISSAT demonstrate how difficult these benchmark problems are for CDCL solvers. With a time limit of 1000 seconds, we found that KISSAT could handle all five instances of Simon's benchmarks with $m = 3$ but none for larger values of m. For Li's benchmarks, it failed for even the minimum case of $m = 3$. Running TBSAT with bucket elimination with a random ordering of the variables fares much better. For Li's benchmarks, it successfully handled instances up to $m = 38$, yielding a proof with around 373 million clauses. For Simon's benchmarks, bucket elimination handled benchmarks for all five seeds up to $m = 60$. We can also see that Simon's benchmarks are decidedly easier than Li's, requiring up to an order of magnitude fewer clauses in the proofs.

Jussila et al. [31] showed benchmark results for what appear to be Simon's Urquhart formulas up to $m = 8$ with performance (in terms of proof size) comparable to ours. Indeed, in using bucket elimination, we are replicating their approach. We know of no prior proof-generating SAT solver that can handle Urquhart formulas of this scale.

5.3 Mutilated Chessboard

The mutilated chessboard problem considers an $n \times n$ chessboard, with the corners on the upper left and the lower right removed. It attempts to tile the board with dominos, with each domino covering two squares. Since the two removed squares had the same color, and each domino covers one white and one black square, no tiling is possible. This problem has been well studied in the context of resolution proofs, for which it can be shown that any proof must be of exponential size [1].

A standard CNF encoding involves defining Boolean variables to represent the boundaries between adjacent squares, set to 1 when a domino spans the two squares, and set to 0 otherwise. The clauses then encode an Exactly1 constraint for each square, requiring each square to share a domino with exactly one of its neighbors. We label the variables representing a horizontal boundary between a square and the one below as $y_{i,j}$, with $1 \leq i < n$ and $1 \leq j \leq n$. The variables representing the vertical boundaries are labeled $x_{i,j}$, with $1 \leq i \leq n$ and $1 \leq j < n$. With a mutilated chessboard, we have that $y_{1,1} = x_{1,1} = y_{n-1,n} = x_{n,n-1} = 0$.

As the plots of Figure 12 show, a straightforward application of linear conjunctions or bucket elimination by TBSAT displays exponential scaling. Indeed, TBSAT fares no better than KISSAT when operating in either of these modes, with all limited to $n \leq 20$ within the 1000-second time limit.

On the other hand, another approach, inspired by symbolic model checking [15], demonstrates far better scaling, reaching $n = 340$. It is based on the following observation: when processing the columns from left to right, the only information required to place dominos in column j is the identity of those rows i for which a domino crosses horizontally from $j - 1$ to j. This information is encoded in the values of $x_{i,j-1}$ for $1 \leq i \leq n$.

In particular, group the variables into columns, with X_j denoting variables $x_{1,j}, \ldots, x_{n,j}$, and Y_j denoting variables $y_{1,j}, \ldots, y_{n-1,j}$. Scanning the board from left to right, consider X_j to encode the

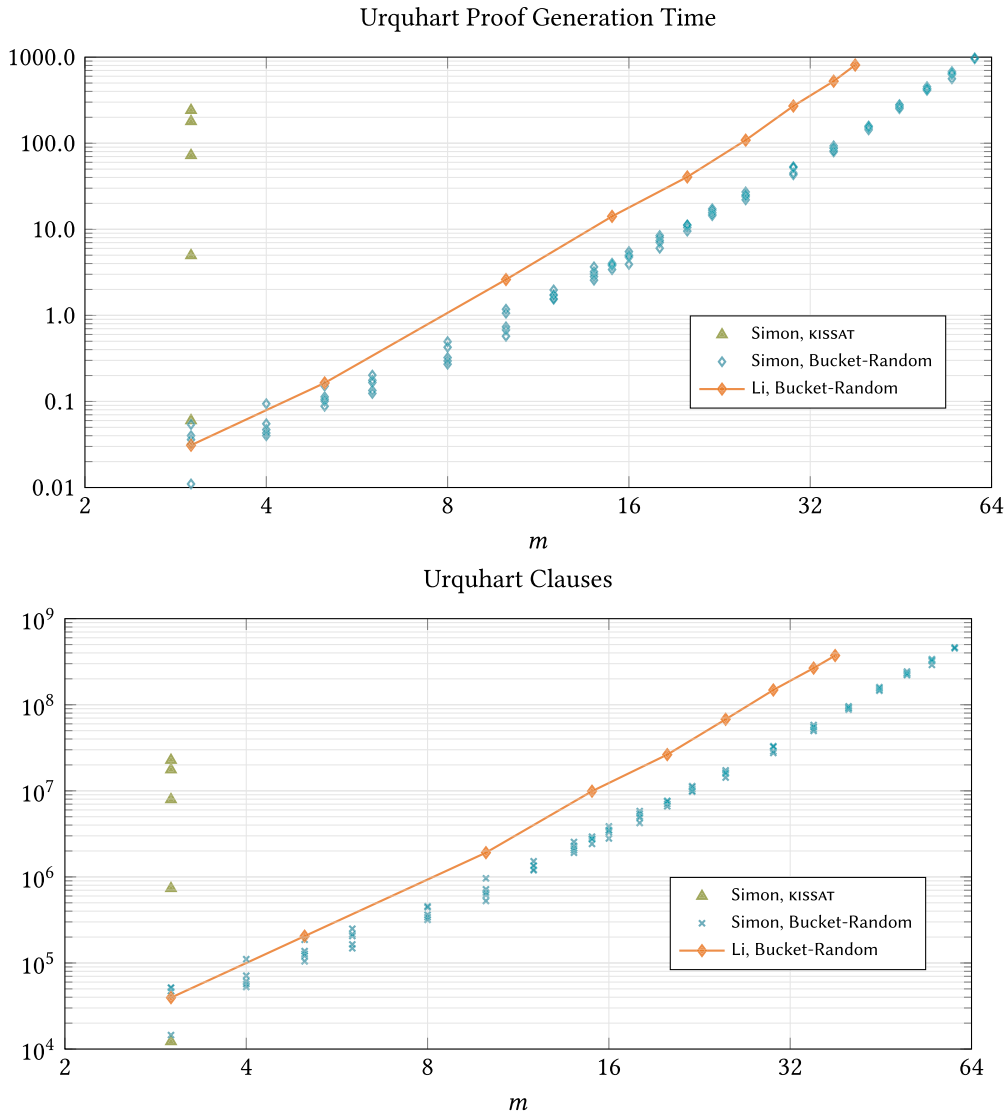


Fig. 11. Generating unsatisfiability proofs for Urquhart formulas with size parameter m. KISSAT timed out for even the minimum-sized version of Li's benchmark ($m = 3$).

"state" of processing after completing column j. As the scanning process reaches column j, there is a *characteristic function* $\sigma_{j-1}(X_{j-1})$ describing the set of allowed crossings of horizontally oriented dominos from column $j-1$ into column j. No other information about the configuration of the board to the left is required. The characteristic function after column j can then be computed as:

$$\sigma_j(X_j) \quad = \quad \exists X_{j-1} \left[\sigma_{j-1}(X_{j-1}) \wedge \exists Y_j \; T_j(X_{j-1}, Y_j, X_j)\right], \tag{3}$$

where $T_j(X_{j-1}, Y_j, X_j)$ is a "transition relation" consisting of the conjunction of the Exactly1 constraints for column j. From this, we can existentially quantify the variables Y_j to obtain a BDD encoding all compatible combinations of the variables X_{j-1} and X_j. By conjuncting this with the

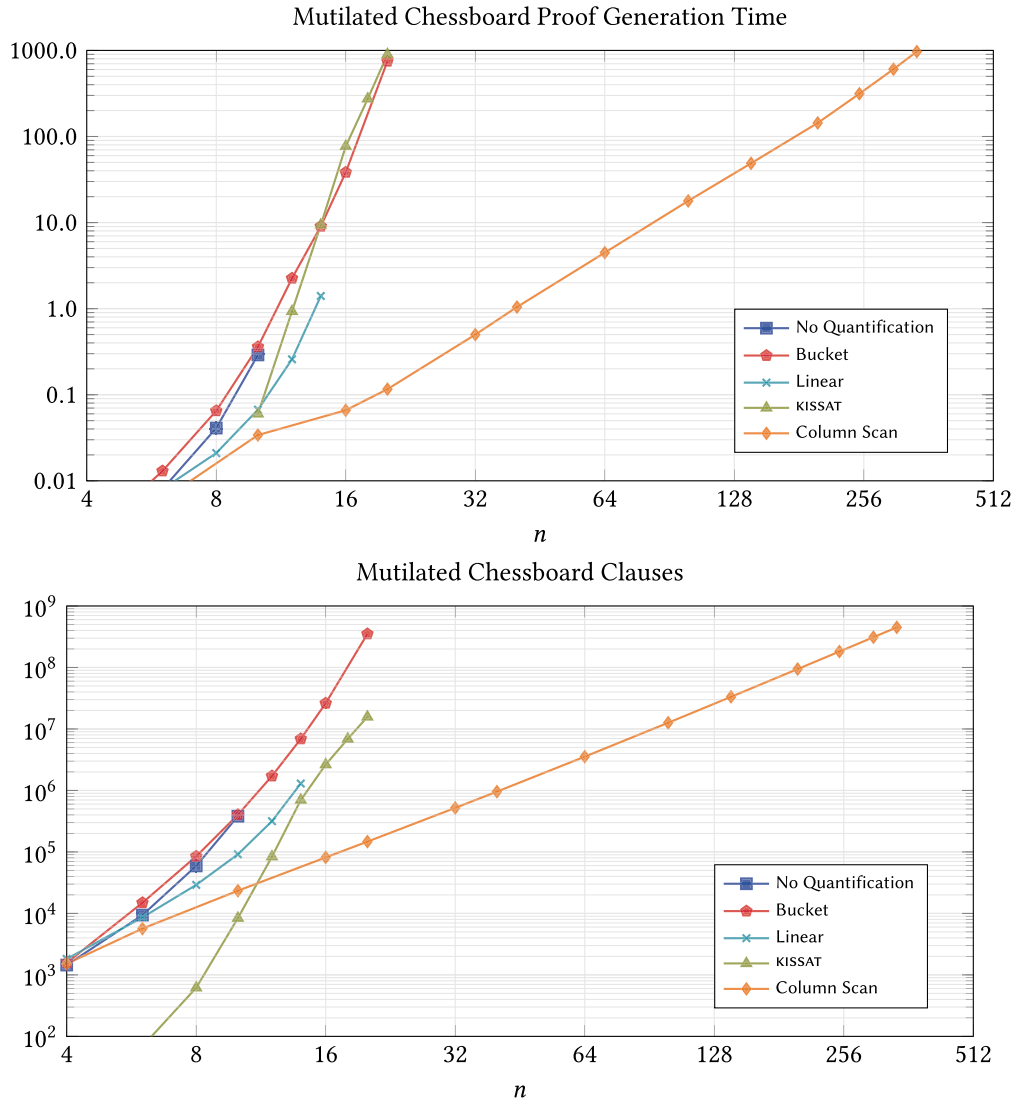


Fig. 12. Generating unsatisfiability proofs for $n \times n$ mutilated chess boards.

characteristic function for column $j-1$ and existentially quantifying the variables X_{j-1}, we obtain the characteristic function for column j. With a mutilated chessboard, we generate leaf node L_0 in attempting the final conjunction. Note that Equation (3) does not represent a reformulation of the mutilated chessboard problem. It simply defines a way to schedule the conjunction and quantification operations over the input clauses.

One important rule-of-thumb in symbolic model checking is that the successive values of the next-state variables must be adjacent in the variable ordering. Furthermore, the vertical variables in Y_j must be close to their counterparts in X_{j-1} and X_j. Both objectives can be achieved by ordering the variables row-wise, interleaving the variables $x_{i,j}$ and $y_{i,j}$, ordering first by row index i and then by column index j. This requires the quantification operations of Equation (3) to be performed on non-root variables.

In our experiments, we found that this scanning reaches a fixed point after processing $n/2$ columns. That is, from that column onward, the characteristic functions become identical, except for a renaming of variables. This indicates that the set of all possible horizontal configurations stabilizes halfway across the board. Moreover, the BDD representations of the states grow as $O(n^2)$. For $n = 340$ largest has just 29,239 nodes. The problem size for the mutilated chessboard scales is n^2, the number of squares in the board. Thus, an instance with $n = 340$ is 289 times larger than an instance with $n = 20$ in terms of the number of input variables and clauses. Column scanning yields a major benefit in the solver performance.

The plot labeled "No Quantification" demonstrates the importance of including existential quantification in solving this problem. These data were generated by using the same schedule as with column scanning but with all quantification operations omitted. As can be seen, this approach could not scale beyond $n = 10$.

It is interesting to reflect on how our column-scanning approach relates to SAT-based bounded model checking (BMC) [4]. This approach to verification encodes the operation of a state transition system for k steps, for some fixed value of k, by instantiating the transition relation k times. It then uses a SAT solver to detect whether some condition can arise within k steps of operation. By contrast, we effectively compress the mutilated chessboard problem into a state machine that adds tiles to successive columns of the board and then perform a BDD-based reachability computation for this system, much as would a symbolic model checker [15]. Just as BDD-based model checking can outperform SAT-based BMC for some problems, we have demonstrated that a BDD-based SAT solver can sometimes outperform a search-based SAT solver.

5.4 Pigeonhole Problem

The pigeonhole problem is one of the most studied problems in propositional reasoning. Given a set of n holes and a set of $n+1$ pigeons, it asks whether there is an assignment of pigeons to holes such that (1) every pigeon is in some hole and (2) every hole contains at most one pigeon. The answer is no, of course, but any resolution proof for this must be of exponential length [27]. Groote and Zantema have shown that any BDD-based proof of the principle that uses only conjunction operations must be of exponential size [26]. On the other hand, Cook constructed an extended resolution proof of size $O(n^4)$, in part to demonstrate the expressive power of extended resolution [18].

We used a representation of the problem that scales as $O(n^2)$, using an encoding of the at-most-one constraints due to Sinz [43]. It starts with a set of variables $p_{i,j}$ for $1 \le i \le n$ and $1 \le j \le n+1$, with the interpretation that pigeon j is assigned to hole i. Encoding the property that each pigeon j is assigned to some hole can be expressed with a single clause:

$$Pigeon_j \quad = \quad \bigvee_{i=1}^{n} p_{i,j}.$$

Sinz's method of encoding the property that each hole i contains at most one pigeon introduces auxiliary variables to effectively track which holes are occupied, starting with pigeon 1 and working upward. These variables are labeled $s_{i,j}$ for $1 \le i \le n$ and $1 \le j \le n$. Informally, variables $s_{i,1}, s_{i,2}, \ldots, s_{i,n}$ serves as a *signal chain* that indicates the point at which a pigeon has been assigned to hole i. For each hole i, there is a total of $3n - 1$ clauses:

Effect	Formula	Clause	Range
Generate	$p_{i,j} \rightarrow s_{i,j}$	$\overline{p}_{i,j}\, s_{i,j}$	$1 \le j \le n$
Propagate	$s_{i,j-1} \rightarrow s_{i,j}$	$\overline{s}_{i,j-1}\, s_{i,j}$	$1 < j \le n$
Suppress	$s_{i,j-1} \rightarrow \overline{p}_{i,j}$	$\overline{s}_{i,j-1}\, \overline{p}_{i,j}$	$1 < j \le n+1$

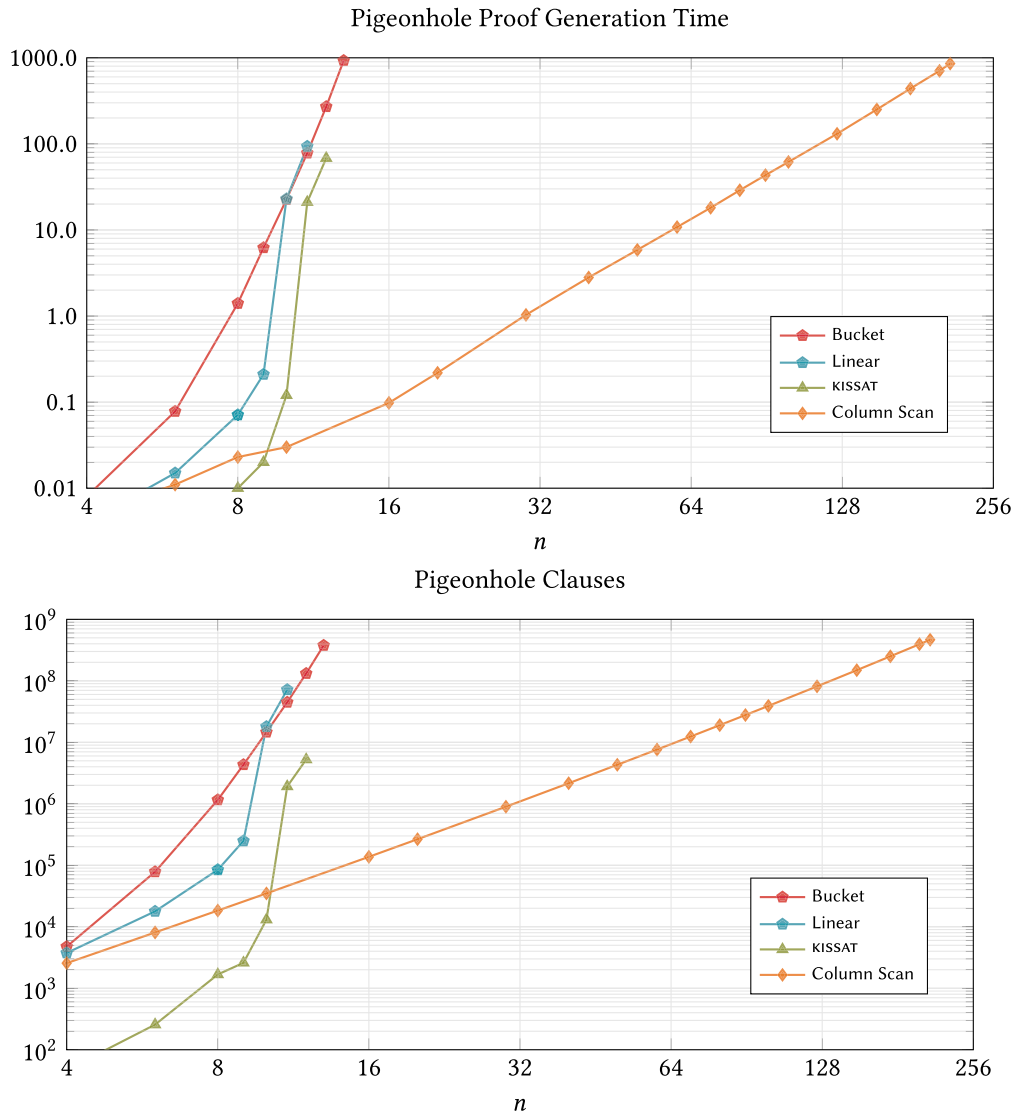


Fig. 13. Generating unsatisfiability proofs for assigning $n + 1$ pigeons to n holes using Sinz's encoding.

Each of these clauses serves either to define how the next value in the chain is to be computed, or to describe the effect of the signal on the allowed assignments of pigeons to the hole. That is, for hole i, the signal is *generated* at position j if pigeon j is assigned to that hole. Once set, the signal continues to *propagate* across higher values of j. Once the signal is set, it *suppresses* further assignments of pigeons to the hole. This encoding requires $3n - 1$ clauses and n auxiliary variables per hole.

Figure 13 shows the results of running the two solvers on this problem. Once again, we see TBSAT with either linear or bucket evaluation having exponential scaling, as does KISSAT. None can go beyond $n = 13$ within the 1000-second time limit.

On the other hand, the column scanning approach used for the mutilated checkerboard can also be applied to the pigeonhole problem when the Sinz encoding is used. Consider an array with hole i

represented by row i and pigeon j represented by column j. Let S_j represent the auxiliary variables $s_{i,j}$ for $1 \leq i \leq n$. The "state" is then encoded in these auxiliary variables. In processing pigeon j, we can assume that the possible combinations of values of auxiliary variables S_{j-1} is encoded by a characteristic function $\sigma_{j-1}(S_{j-1})$. In addition, we incorporate into this characteristic function the requirement that each pigeon k, for $1 \leq k \leq j-1$ is assigned to some hole. Letting P_j denote the variables $p_{i,j}$ for $1 \leq i \leq n$, the characteristic function at column j can then be expressed as

$$\sigma_j(S_j) \quad = \quad \exists S_{j-1} \left[\sigma_{j-1}(S_{j-1}) \wedge \exists P_j \; T_j(S_{j-1}, P_j, S_j)\right], \tag{4}$$

where the "transition relation" T_j consists of the clauses associated with the auxiliary variables plus the clause encoding constraint $Pigeon_j$. As with the mutilated chessboard, having a proper variable ordering is critical to the success of a column scanning approach. We interleave the ordering of the variables $p_{i,j}$ and $s_{i,j}$, ordering them first by i (holes) and then by j (pigeons).

Figure 13 demonstrates the effectiveness of the column-scanning approach. We were able to handle instances up to $n = 210$. Unlike with the mutilated chessboard, the scanning does not reach a fixed point. Instead, the BDDs start very small, because they must encode the locations of only a small number of occupied holes. They reach their maximum size at pigeon $n/2$, as the number of combinations for occupied and unoccupied holes reaches its maximum of $C(n, n/2)$. The BDD sizes then drop off, symmetrically to the first $n/2$ pigeons, as the encoding needs to track the positions of a decreasing number of unoccupied holes. Fortunately, all of these BDDs scale quadratically with n, reaching a maximum of 11,130 nodes for $n = 210$.

We also ran experiments using a *direct* encoding of the at-most-one constraints, having a clause $\overline{p}_{i,j} \vee \overline{p}_{i,k}$ for each hole i and for $1 \leq j < k \leq n+1$. This encoding scales as $\Theta(n^3)$. With this encoding, we were unable to find any method that avoided exponential scaling using either TBSAT or KISSAT.

5.5 Evaluation

Overall, our results demonstrate the potential for generating small proofs of unsatisfiability using BDDs. We were able to greatly outperform traditional CDCL solvers for four well-known challenge problems.

The success for the first two benchmark problems relies on the ability of BDDs to handle exclusive-or operations efficiently. Generally, the exclusive-or of k variables can be expressed as a BDD with $2k + 1$ nodes, including the leaves. These representations are also independent of the variable ordering. As we saw, however, it is critical to quantify variables whenever possible to avoid requiring the BDD to encode the parity relationships among many overlapping subsets of the variables. We found that bucket elimination works well on these problems and that randomness in the problem structure and the variable ordering did not adversely affect performance. This strategy was outlined by Jussila et al. [31]; our experimental results serve as a demonstration of the utility of their work.

The success of column scanning for the final two benchmark problems relies on finding a way to scan in one dimension, encoding the "state" of the scan in a compact form. This strategy only works when the problem is encoded in a way that it can be partitioned along two dimensions. This approach draws its inspiration from symbolic model checking, and it requires the more general capability to handle quantification that we have presented. One strength of modern SAT solvers is that they generally succeed without any special guidance from the user. It remains an open question whether column scanning can be made more general and whether a suitable schedule and variable ordering can be generated automatically. Without these capabilities, our results for column scanning show promise, but they require too much guidance from the user.

Table 1. Summary Data for the Largest Parity and Urquhart Formulas Solved

Instance	Parity-9750	Urquhart-Li-38
Input variables	29,244	29,868
Input clauses	77,984	79,648
Total BDD nodes	62,722,228	55,763,704
Total clauses	419,255,800	372,999,366
Maximum live clauses	166,706,941	148,101,720
Solver time (secs)	928.4	809.5
Checking time (secs)	288.3	258.5

Table 2. Summary Data for the Largest Chess and Pigeonhole Problems Solved

Instance	Chess-340	Pigeon-Sinz-210
Input variables	230,516	88,410
Input clauses	805,112	132,301
Total BDD nodes	100,804,928	53,093,749
Total clauses	449,676,065	465,887,970
Maximum live clauses	119,957,540	30,295,942
Solver time (secs)	969.4	857.2
Checking time (secs)	298.0	340.4

Other studies have compared BDDs to CDCL solvers on a variety of benchmark problems. Several of these observed exponential performances for BDD-based solvers for problems for which we have obtained more promising results. Uribe and Stickel [47] ran experiments with the mutilated chessboard problem, but they did not do any variable quantification. Pan and Vardi [39] applied a variety of scheduling and variable ordering strategies for the mutilated chessboard and pigeonhole problems. Although they found that they could get better performance than with a CDCL solver, their performance still scaled exponentially. Obtaining scalability requires devising more problem-specific approaches than the ones they considered. Our experiments with KISSAT confirm that a BDD-based SAT solver requires careful attention to the problem encoding, the variable ordering, and the use of quantification in order to outperform a state-of-the CDCL solver.

Tables 1 and 2 provide some performance data for the largest instances solved for each of the four benchmark problems. A first observation is that these problems are very large, with tens of thousands of input variables and clauses.

Looking at the BDD data, the total number of BDD nodes indicates the total number generated by the function GETNODE and for which extension variables are created. These are numbered in the millions, and far exceed the number of input variables.

The entries for "Maximum live clauses" show the peak number of clauses that had been added but not yet deleted across the entire proof. As can be seen, these can vary from 7% to nearly 40% of the total clauses. The peak number of live clauses proved to be a limiting factor for the LRAT proof checker.

Figure 14 provides more insight into the nature of the proofs generated by the CDCL solver KISSAT and the BDD-based solver TBSAT. Each point indicates one benchmark run, with the value on the Y axis indicating the runtime of the solver divided by the number of clauses generated, scaled by 10^6, whereas the X value is the proof size. In other words, the Y values show the average

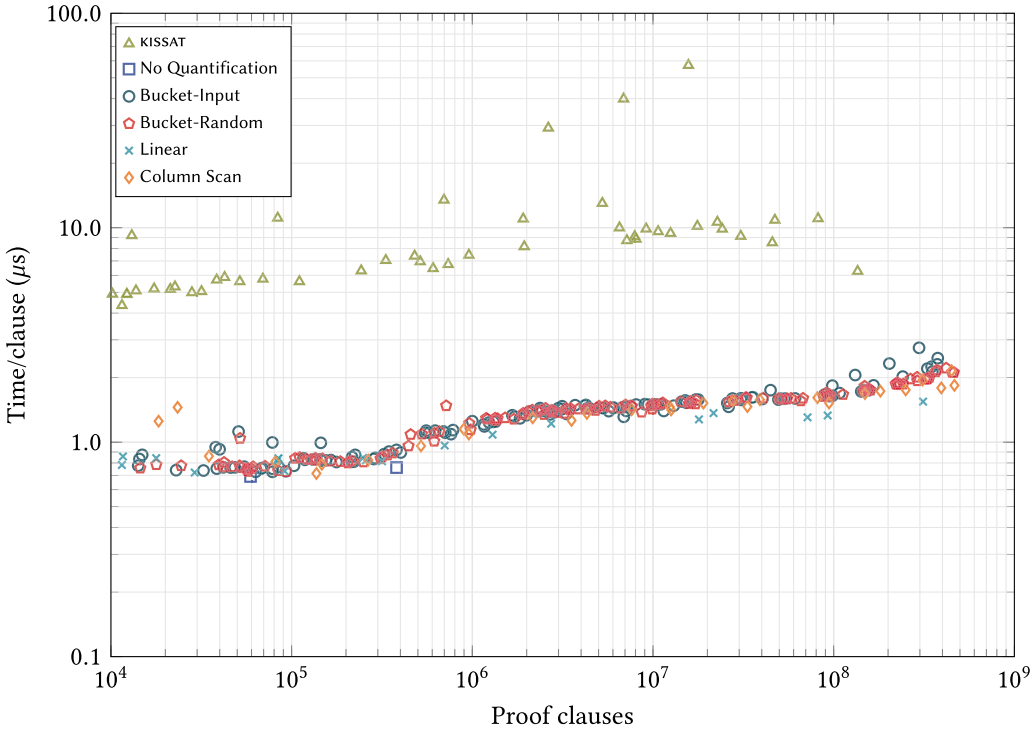


Fig. 14. Average generation time (μs) per clause across all benchmarks.

time in microseconds for each proof clause to be generated. In all, 340 points are shown, with 75 for KISSAT and the rest for TBSAT in its various operating modes.

These data reflect a fundamental difference between how proofs are generated with a CDCL solver versus with a BDD-based solver. A CDCL solver emits a clause each time it encounters a conflict during the search. This may come after many steps involving selecting a decision variable and performing Boolean constraint propagation. Thus, there can be considerable and highly variable amounts of processing between successive clause emissions. We see average times ranging between 4 and 60 microseconds for the KISSAT runs, and even these averages mask the considerable variations that can occur within a single run.

With a BDD-based solver, on the other hand, the proof has the form of a log describing the recursive steps taken by the BDD algorithm, expressed within a standard proof framework. There is very little variability from one run to the next, and the different evaluation modes have minimal impact. The only trend of note is a general increase in the average time per clause as the proofs get longer. The short runs require less than 1.0 microsecond per clause, whereas the longer ones require over 2.0. This increase can be attributed to the complexity of managing long BDD computations, requiring garbage collection, table resizing, and other overhead operations.

Figure 15 shows a similar plot, but with the Y axis indicating the average time for the proof checkers to check each clause. Again, we see two important characteristics. The proof steps generated by KISSAT do not include lists of antecedent clauses (hints). Instead, the checking program DRAT-TRIM scans the set of clauses and constructs each hint sequence. This takes significant effort and can vary greatly across benchmarks. The proofs generated by TBSAT, on the other hand, contain full hints and can therefore be readily checked at an average of around 0.7 μ seconds per proof clause, regardless of the proof size or solution method.

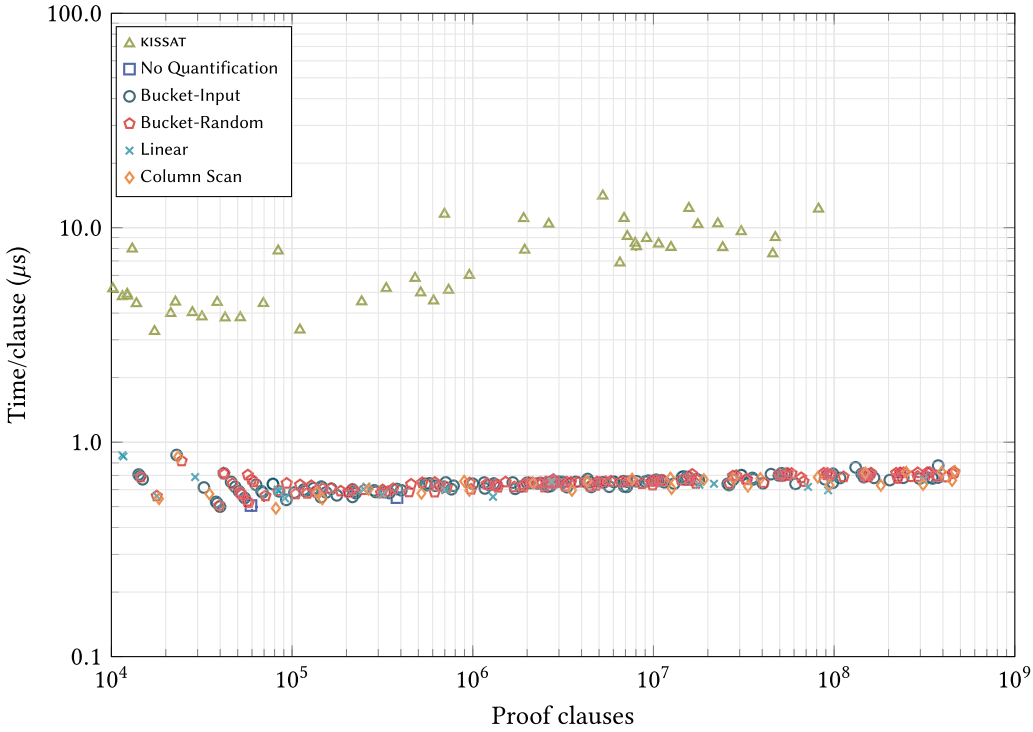


Fig. 15. Average checking time (μs) per clause across all benchmarks.

6 CONCLUSION

The pioneering work by Biere, Sinz, and Jussila [31, 44] did not lead to as much follow-up work as it deserved. Here, many years later, we found that small modifications to their approach enable a powerful BDD-based SAT solver to generate proofs of unsatisfiability. The key to its success is the ability to perform arbitrary existential quantification. As the experimental results demonstrate, such a capability is critical to obtaining reasonable performance.

More advanced BDD-based SAT solvers employ additional techniques to improve their performance. Extending our methods to handle these techniques would be required to have them generate proofs of unsatisfiability. Some of these would be straightforward. For example, Weaver et al. [50] derive a very general set of conditions under which existential quantification can be applied while preserving satisfiability. For generating proofs of unsatisfiability, our ability to prove that existential quantification preserves implication would be sufficient for all of these cases. On the other hand, more advanced solvers, such as SBSAT [23], employ a variety of techniques to prune the intermediate BDDs based on the structure of other BDDs that remain to be conjuncted. This pruning generally reduces the set of satisfying assignments to the BDD; thus implication does not hold.

In more recent work, we have been able to show that BDD-based methods can use solution methods that view a Boolean formula as encoding linear equations over integers or modular integers [10]. Proof-generating BDD operations can be used to justify the individual steps taken while solving systems of equations by several different methods. That has allowed us to scale the benchmark problems considered in Section 5 even further, and to avoid the need for problem-specific solution methods. We have also demonstrated that proof-generating BDDs can be integrated into

a convention CDCL solver to allow it to use Gauss-Jordan elimination on the parity constraints encoded in the formula [13]. Overall, we believe that BDD-based methods can augment other SAT solving methods to provide new capabilities.

The ability to generate correctness proofs in a BDD-based SAT solver invites us to also consider generating proofs for other tasks to which BDDs are applied. We have already done so for quantified Boolean formulas, demonstrating the ability to generate proofs for both true and false formulas in a unified framework [11]. Other problems of interest include model checking and model counting. Perhaps a proof of unsatisfiability could provide a useful building block for constructing correctness proofs for these other tasks.

ACKNOWLEDGMENTS

We would like to thank Chu-Min Li and Laurent Simon for sharing their programs for generating the two classes of Urquhart benchmarks evaluated in Section 5.2.

REFERENCES

[1] Michael Alekhnovich. 2004. Mutilated chessboard problem is exponentially hard for resolution. *Theoretical Computer Science* 310, 1-3 (Jan. 2004), 513–525.

[2] Henrik Reif Andersen. 1997. *An Introduction to Binary Decision Diagrams*. Technical Report. Technical University of Denmark.

[3] Armin Biere. 2019. CaDiCaL at the SAT Race 2019. In *Proc. of SAT Race 2019 – Solver and Benchmark Descriptions (Department of Computer Science Series of Publications B)*, Vol. B-2019-1. University of Helsinki, 8–9.

[4] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99) (LNCS)*, Vol. 1579. 193–207.

[5] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. 2020. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020. In *Proc. of SAT Competition 2020—Solver and Benchmark Descriptions (Department of Computer Science Report Series B)*, Vol. B-2020-1. University of Helsinki, 51–53.

[6] Randal E. Bryant. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers* 35, 8 (1986), 677–691.

[7] Randal E. Bryant. 1992. Symbolic Boolean manipulation with ordered binary decision diagrams. *Comput. Surveys* 24, 3 (September 1992), 293–318.

[8] Randal E. Bryant. 2018. Binary decision diagrams. In *Handbook of Model Checking*, Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer, 191–217.

[9] Randal E. Bryant. 2022. TBUDDY: A proof-generating BDD package. In *Formal Methods in Computer-Aided Design (FMCAD'22)*.

[10] Randal E. Bryant, Armin Biere, and Marijn J. H. Heule. 2022. Clausal proofs for pseudo-Boolean reasoning. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'22) (LNCS)*, Vol. 12651. 76–93.

[11] Randal E. Bryant and Marijn J. H. Heule. 2021. Dual proof generation for quantified Boolean formulas with a BDD-based solver. In *Conference on Automated Deduction (CADE'21) (LNAI)*, Vol. 12699. 433–449.

[12] Randal E. Bryant and Marijn J. H. Heule. 2021. Generating extended resolution proofs with a BDD-based SAT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part I (LNCS)*, Vol. 12651. 76–93.

[13] Randal E. Bryant and Mate Soos. 2022. Proof Generation for CDCL Solvers Using Gauss-Jordan Elimination. (2022).

[14] Jerry R. Burch, Edmund M. Clarke, and David E. Long. 1991. Symbolic model checking with partitioned transition relations. In *VLSI91*.

[15] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. 1992. Symbolic model checking: 10^{20} states and beyond. *Information and Computation* 98, 2 (1992), 142–170.

[16] Philippe Chatalic and Laurent Simon. 2000. ZRes: The old Davis-Putnam procedure meets ZBDD. In *Conference on Automated Deduction (CADE'00) (LNCS)*, Vol. 1831. 449–454.

[17] Leroy Chew and Marijn J. H. Heule. 2020. Sorting parity encodings by reusing variables. In *Theory and Applications of Satisfiability Testing (SAT'20) (LNCS)*, Vol. 12178. 1–10.

[18] Stephen A. Cook. 1976. A short proof of the pigeon hole principle using extended resolution. *SIGACT News* 8, 4 (Oct. 1976), 28–32.

[19] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. 2017. Efficient certified RAT verification. In *Conference on Automated Deduction (CADE'17) (LNCS)*, Vol. 10395. 220–236.

[20] Luís Cruz-Filipe, João P. Marques-Silva, and Peter Schneider-Kamp. 2017. Efficient certified resolution proof checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (LNCS)*, Vol. 10205. 118–135.
[21] Martin Davis and Hilary Putnam. 1960. A computing procedure for quantification theory. *J. ACM* 7, 3 (1960), 201–215.
[22] Rina Dechter. 1999. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence* 113, 1–2 (1999), 41–85.
[23] John Franco, Michal Kouril, John Schlipf, Jeffrey Ward, Sean Weaver, Michael Dransfield, and W. Mark Vanfleet. 2004. SBSAT: A state-based, BDD-based satisfiability solver. In *Theory and Applications of Satisfiability Testing (SAT) (LNCS)*, Vol. 2919. 398–410.
[24] Ofer Gabber and Zvi Galil. 1981. Explicit construction of linear-sized superconcentrators. *J. Comput. System Sci.* 22 (1981), 407–420.
[25] Evgueni I. Goldberg and Yakov Novikov. 2003. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe (DATE'03)*. 886–891.
[26] Jan F. Groote and H. Zantema. 2003. Resolution and binary decision diagrams cannot simulate each other polynomially. *Discrete Applied Mathematics* 130, 2 (2003), 157–171.
[27] Armin Haken. 1985. The intractability of resolution. *Theoretical Computer Science* 39 (1985), 297–308.
[28] Marijn J. H. Heule and Armin Biere. 2015. Proofs for satisfiability problems. In *All about Proofs, Proofs for All (APPA'15)*. Math. Logic and Foundations, Vol. 55. College Pub.
[29] Marijn J. H. Heule, Warren A. Hunt, Jr., and Nathan D. Wetzler. 2013. Verifying refutations with extended resolution. In *Conference on Automated Deduction (CADE'13) (LNCS)*, Vol. 7898. 345–359.
[30] Shlomo Hoory, Nathan Linial, and Avi Wigderson. 2006. Expander graphs and their applications. *Bull. Amer. Math. Soc.* 43, 4 (October 2006), 439–561.
[31] Toni Jussila, Carsten Sinz, and Armin Biere. 2006. Extended resolution proofs for symbolic SAT solving with quantification. In *Theory and Applications of Satisfiability Testing (SAT'06) (LNCS)*, Vol. 4121. 54–60.
[32] Oliver Kullmann. 1999. On a generalization of extended resolution. *Discrete Applied Mathematics* 96-97 (1999), 149–176.
[33] Peter Lammich. 2020. Efficient verified (UN)SAT certificate checking. *Journal of Automated Reasoning* 64 (2020), 513–532.
[34] Chu-Min Li. 2003. Equivalent literal propagation in the DLL procedure. *Discrete Applied Mathematics* 130, 2 (2003), 251–276.
[35] Gregori A. Margulis. 1973. Explicit construction of concentrators. *Probl. Perdachi Info (Problems in Information Transmission)* 9, 4 (1973), 71–80.
[36] João Marques-Silva, Ines Lynce, and Sharad Malik. 2009. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*. IOS Press, 131–153.
[37] Donald Michie. 1968. 'Memo' functions and machine learning. *Nature* 218 (1968), 19–22.
[38] Shin-Ichi Minato, Nagisa Ishiura, and Shuzo Yajima. 1990. Shared binary decision diagrams with attributed edges for efficient Boolean function manipulation. In *Design Automation Conference (DAC'90)*. 52–57.
[39] Guoqiang Pan and Moshe Y. Vardi. 2005. Search vs. symbolic techniques in satisfiability solving. In *Theory and Applications of Satisfiability Testing (SAT'05) (LNCS)*, Vol. 3542. 235–250.
[40] Tobias Philipp and Adrián Rebola-Pardo. 2017. Towards a semantics of unsatisfiability proofs with inprocessing. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'17)*. 65–84.
[41] John A. Robinson. 1965. A machine-oriented logic based on the resolution principle. *J. ACM* 12, 1 (January 1965), 23–41.
[42] Laurent Simon, Daniel Le Beurre, and Edward A. Hirsch. 2005. The SAT2002 competition. *Annals of Mathematics and Artificial Intelligence* 43 (2005), 307–342.
[43] Carsten Sinz. 2005. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Principles and Practice of Constraint Programming (CP'05) (LNCS)*, Vol. 3709. 827–831.
[44] Carsten Sinz and Armin Biere. 2006. Extended resolution proofs for conjoining BDDs. In *Computer Science Symposium in Russia (CSR'06) (LNCS)*, Vol. 3967. 600–611.
[45] Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. 2021. cake_lpr: Verified propagation redundancy checking in CakeML. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part II (LNCS)*, Vol. 12652. 223–241.
[46] Gregori S. Tseitin. 1983. On the complexity of derivation in propositional calculus. In *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Springer, 466–483.
[47] Tomás E. Uribe and Mark E. Stickel. 1994. Ordered binary decision diagrams and the Davis-Putnam procedure. In *Constraints in Computational Logics (LNCS)*, Vol. 845. 34–49.
[48] Alasdair Urquhart. 1995. The complexity of propositional proofs. *The Bulletin of Symbolic Logic* 1, 4 (1995), 425–467.

[49] Allen Van Gelder. 2012. Producing and verifying extremely large propositional refutations. *Annals of Mathematics and Artificial Intelligence* 65, 4 (2012), 329–372.
[50] Sean Weaver, John V. Franco, and John S. Schlipf. 2006. Extending existential quantification in conjunctions of BDDs. *JSAT* 1, 2 (2006), 89–110.
[51] Nathan D. Wetzler, Marijn J. H. Heule, and Warren A. Hunt Jr. 2014. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Theory and Applications of Satisfiability Testing (SAT'14) (LNCS)*, Vol. 8561. 422–429.
[52] Lintao Zhang and Sharad Malik. 2003. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe (DATE'03)*. 880–885.

Received 30 April 2021; revised 27 March 2023; accepted 25 April 2023