Trimming while Checking Clausal Proofs

Marijn J.H. Heule, Warren A. Hunt, Jr., and Nathan Wetzler
The University of Texas at Austin

Abstract—**Conflict-driven clause learning (CDCL) satisfiability solvers can emit more than a satisfiability result; they can also emit clausal proofs, resolution proofs, unsatisfiable cores, and Craig interpolants. Such additional results may require substantial modifications to the solver, especially if preprocessing and inprocessing techniques are used; however, CDCL solvers can easily emit clausal proofs with very low overhead. We present a new approach with an associated tool that efficiently validates clausal proofs and can distill additional results from clausal proofs. Our tool architecture makes it easy to obtain such results from any CDCL solver. Experimental evaluation shows that our tool can validate clausal proofs faster than existing tools. Additionally, the quality of the additional results, such as unsatisfiable cores, is higher when compared to modified SAT solvers.**

I. Introduction

Conflict-driven clause learning (CDCL) satisfiability solvers compute the satisfiability of a given Boolean formula. When a solver claims a formula is unsatisfiable, most solvers can also emit a proof of unsatisfiability as a sequence of learned clauses and some solvers can procuce an unsatisfiable core of the clauses used to refute a formula. Such proofs can then be checked to validate the unsatisfiability claim of a CDCL solver, while the core can be used as a starting point for extracting minimal unsatisfiable subsets (MUS) and interpolants.

Proofs of unsatisfiability can be expressed in clausal- or resolution-style formats [1], [2], [3], [4], and such proofs provide assurance that a solver is correct [5]. Any CDCL solver can emit clausal proofs with low overhead, and clausal proofs are much smaller than resolution-style proofs. However, clausal proofs are relatively expensive to validate, and clausal checkers can be complicated, making them harder to trust or mechanically verify. Although resolution proofs are easy to validate with a simple proof checker, they are hard to obtain and can be huge in size. This paper provides additional evidence that clausal proofs are more useful in practice.

SAT solvers that emit additional results [6], such as unsatisfiable cores, store an *antecedent graph*, a directed acyclic graph that represents a dependency between learned and input clauses. Storing an antecedent graph requires significant modifications to a SAT solver's implementation. The size of the antecedent graph can be two orders of magnitude larger than the size of the clause database. This requires a lot of memory, even with optimizations [7], and can slow down a solver significantly. Also, it may be hard to compute the antecedents of learned clauses for some reasoning techniques, such as equivalence reasoning and hidden literal elimination [8]. Few solvers support the storing of antecedent graphs, and no top-tier SAT solver has the ability to store them.

This paper uses clausal proof checking to produce additional results from SAT solvers. More specifically, we reconstruct an antecedent graph from a clausal proof rather than producing it while solving. Goldberg and Novikov [1] proposed an algorithm, known as *backward checking*, to achieve this. As far as we know, there is no implementation of this method available. We noticed that the method described in [1] can be very expensive when used on clausal proofs from state-of-the-art CDCL solvers. In this paper, we present two optimizations for this algorithm to reduce its computational costs substantially: we add clause deletion information to clausal proofs and we develop an alternative procedure to perform unit propagation.

We have implemented a proof-checking tool, called DRUP-trim, that mitigates one of the main drawbacks of clausal proof checking, namely speed[1]. CDCL SAT solvers can easily emit clausal proofs, and these proofs can now be used to produce additional results from any SAT solver. Our DRUP-trim tool also enables validation of lookahead SAT solvers [9]. These solvers use several types of *local learning* that make it hard to emit resolution proofs; however, clausal proofs are easy to emit. Our work is most closely related to that of Van Gelder [3] whose RUP2RES tool converts clausal proofs into resolution proofs. In contrast to RUP2RES, our DRUP-trim tool can emit additional results such as unsatisfiable cores and reduced proofs. DRUP-trim does not store arcs in the antecedent graph and therefore does not suffer from high memory consumption.

Our contributions are in three areas: verification of unsatisfiability proofs, minimal unsatisfiable core extraction, and computation of Craig interpolants. Our DRUP-trim tool facilitates fast validation of unsatisfiability results of CDCL solvers and, in the process, generates additional results that can be used as a starting point for tools that produce MUSes or Craig interpolants. Most preprocessing techniques used in state-of-the-art CDCL solvers [10] can be easily converted into clausal proofs which can be used to obtain additional results for problems that are too hard to solve without them.

Our paper begins with an introduction to satisfiability, resolution, Boolean constraint propagation, and clausal proofs in Section II. In Section III, we review antecedent graphs and their applications and optimizations. Next, we present a series of improvements to clausal proof checking: backward reverse unit propagation (Section IV), the addition of clause deletion information (Section V), and a preference for clauses that are already marked as part of the core (Section VI). In Section VII, we evaluate our method and we conclude in Section VIII.

[1]A slightly modified version of this tool will be used to validate the unsatisfiability results of SAT Competition 2013.

II. Preliminaries

We briefly review necessary background concepts: conjunctive normal form (CNF), resolution, Boolean constraint propagation, and clausal proofs.

A. Conjunctive Normal Form

For a Boolean variable x, there are two *literals*, the positive literal, denoted by x, and the negative literal, denoted by $\bar{x}$. A *clause* is a finite disjunction of literals, and a CNF *formula* is a finite conjunction of clauses. The set of literals occurring in a CNF formula F is denoted by $\mathrm{LIT}(F)$. A truth assignment for a CNF formula F is a partial function τ that maps literals $l \in \mathrm{LIT}(F)$ to $\{\mathbf{t}, \mathbf{f}\}$. If $\tau(l) = v$, then $\tau(\bar{l}) = \neg v$, where $\neg \mathbf{t} = \mathbf{f}$ and $\neg \mathbf{f} = \mathbf{t}$. An assignment can also be thought of as a conjunction of literals. Furthermore, given an assignment τ:

- A clause C is *satisfied* by τ if $\tau(l) = \mathbf{t}$ for some $l \in C$.
- A clause C is *falsified* by τ if $\tau(l) = \mathbf{f}$ for all $l \in C$.
- A formula F is *satisfied* by τ if $\tau(C) = \mathbf{t}$ for all $C \in F$.
- A formula F is *falsified* by τ if $\tau(C) = \mathbf{f}$ for some $C \in F$.

A CNF formula with no satisfying assignments is called *unsatisfiable*. A clause C is *logically implied* by formula F if adding C to F does not change the set of satisfying assignments of F.

B. Resolution

The resolution rule states that, given two clauses $C_1 = (x \vee a_1 \vee \ldots \vee a_n)$ and $C_2 = (\bar{x} \vee b_1 \vee \ldots \vee b_m)$, the clause $C = (a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)$, can be inferred by resolving on variable x. We say C is the *resolvent* of C_1 and C_2, while C_1 and C_2 are the *antecedents* of C. We write $C = C_1 \bowtie C_2$. The resolvent C is logically implied by any formula containing C_1 and C_2.

C. Boolean Constraint Propagation

A clause C is *unit* under an assignment τ if (1) there exists exactly one literal $l \in C$ such that $l \notin \tau$ and $\bar{l} \notin \tau$, and (2) for all $l' \in C$ such that $l' \neq l$, $\bar{l'} \in \tau$. We say l is the *unit literal* for *unit clause* C. Given a formula F and an assignment τ, *Boolean constraint propagation* $\mathrm{BCP}(F, \tau)$, also known as *unit propagation*, repeatedly extends τ with unit literals (for a unit clause $C \in F$ under τ) until a fixed point is achieved. If there exists an l such that $l \in \tau$ and $\bar{l} \in \tau$, we say that BCP *derives a conflict*.

Example 1. Given the formula $F = (a \vee b \vee c) \wedge (a \vee \bar{b})$ and the assignment $\tau = (\bar{a})$, BCP extends τ with unit literal $\bar{b}$ and then with unit literal c. As a result, $\mathrm{BCP}(F, \tau) = (\bar{a} \wedge \bar{b} \wedge c)$.

D. Clausal Proofs

Goldberg and Novikov [1] introduced *clausal proofs* as an alternative to resolution-style proofs [2] of unsatisfiability. They observed that each clause learned by CDCL conflict analysis can be validated using BCP. Learned clauses are disjunctions of literals, and the complement of a clause, written $\bar{C}$, can be interpreted as an assignment. If $\mathrm{BCP}(F, \bar{C})$ derives a conflict, then C is logically implied by F. This process is also known as *reverse unit propagation* (RUP) [3]. A clausal proof, then, consists of a sequence of learned clauses that have the *RUP property*; i.e., they can be validated using RUP. A (clausal) refutation is a proof that contains the empty clause.

In order to distinguish learned clauses from input clauses, we appeal to the notion that *lemmas* are used to construct a proof of a theorem. Here, learned clauses are lemmas which support a theorem stating that a formula is unsatisfiable. From now on, we will use the term clauses to refer to input clauses, while lemmas will refer to learned clauses.

The elegance of clausal proofs is that they can be expressed in conjunctive normal form; however, the order of lemmas in the proof is important. Clausal proofs are significantly smaller when compared to resolution proofs, and only minor modifications of a SAT solver are required to output clausal proofs. However, clausal proof checking can be quite expensive. And, checking algorithms for clausal proofs are also typically more complex than those for resolution proofs, making it harder to trust or prove correctness of the algorithm.

III. Antecedent Graphs

An *antecedent graph* is a directed acyclic graph that represents the refutation of a formula. The root nodes of an antecedent graph represent the clauses in the original formula, and internal nodes represent lemmas. A directed arc from node C_1 to node C_2 signifies the use of C_1 in the construction of C_2. In other words, C_1 is an antecedent for C_2. One of the leaf nodes in the antecedent graph is the empty clause.

Example 2. Consider the formula $(\bar{b} \vee c) \wedge (a \vee c) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee \bar{b}) \wedge (a \vee \bar{b}) \wedge (b \vee \bar{c})$. Fig. 1 shows an antecedent graph of this formula in which clauses are shortened: $\bar{b}c$ for $(\bar{b} \vee c)$. The antecedent graph consists of four lemmas including the empty clause. For each lemma, the set of incoming arcs represents the antecedents. So the antecedents of lemma (c) are clauses $(\bar{b} \vee c)$, $(a \vee c)$, and $(\bar{a} \vee b)$.

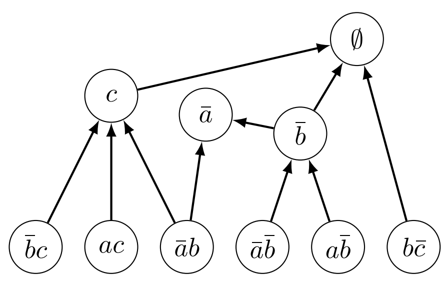

Fig. 1. Antecedent graph for an example formula. Apart from the lemma $(\bar{a})$, all clauses and lemmas are in the cone of $\emptyset$.

The *cone* of a lemma L is the set of all clauses and lemmas from which L is reachable. We refer to *core clauses* as the clauses that are in the cone of the empty clause. Similarly, *core lemmas* refer to the lemmas in the cone of the empty clause and *core arcs* are all incoming arcs of core lemmas. If a solver stores the antecedent graph, it is easy to compute the core clauses, lemmas, or arcs by simply checking for reachability to the empty clause.

The number of core arcs is typically 300 to 400 times larger than the number of core lemmas. To illustrate this difference, we computed the number of core arcs and core lemmas using Picosat [6] while solving the application benchmarks of the SAT 2009 suite, the results of which are shown as a scatter plot in Fig. 2. See Section VII for the details of the machine used in this experiement. Compared to the number of literals in core lemmas, the number of core arcs is about 10 times larger. That means that the memory consumption of a solver that stores the antecedent graph is at least 10 times larger as compared to a solver that does not store the full graph. In practice, this number can be significantly larger because the solver needs to keep some deleted lemmas — even with optimizations [7]. We observed that emitting additional results by Picosat (which requires an antecedent graph) increased the memory requirement by a factor 100 for several benchmarks. Consequently, storing the antecedent graph while solving can reduce the performance of solvers significantly and results in memory exhaustion.

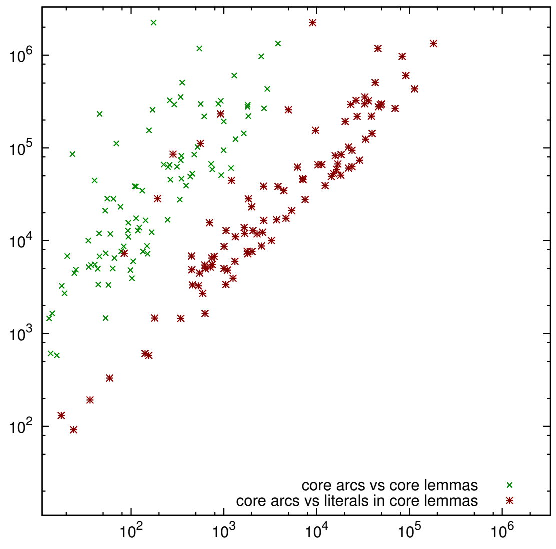

Fig. 2. A scatter plot of the number of core arcs (y-axis) versus the number of core lemmas (green) and the number of literals in core lemmas (red).

In this paper, we propose to reconstruct the antecedent graph after the search has ended by using clausal proofs. Hence, SAT solvers that want to compute additional results will no longer need to store arcs in the antecedent graph.

Applications

Verification of unsatisfiability proofs: The tool that we present in this paper is a fast proof checker for clausal unsatisfiability proofs. Our tool allows developers and users to verify solver output. Additionally, our tool can emit a reduced proof with optimal clause deletion information. A reduced clausal proof might then be be validated by a mechanically-verified proof checker.

Minimal unsatisfiable core extraction: Computing a minimal unsatisfiable subset (MUS) consists of two phases. In the first phase, called *trimming*, the input formula is solved and an antecedent graph is constructed. All original and learned clauses that are not in the cone of the empty clause are removed. This phase can substantially reduce the number of original and learned clauses, and this phase can be repeated. The second phase [11] repeats the following until a fixed point is reached. Select an original clause that is not marked as a clause in the minimal unsatisfiable core. Next, a new Boolean formula is constructed that consists of the remaining original clauses (without the selected clause) and all learned clauses that do not have the selected clause in their dependency cone. If this Boolean formula is satisfiable, the selected clause is marked as part of the minimal unsatisfiable core; otherwise, the selected clause and all learned clauses that have this clause in their dependency cone are removed.

Computing Craig interpolants.: Another application from the field of model checking relies on the availability of an antecedent graph to compute *Craig interpolants* [12]. Given two satisfiable Boolean formulas A and B such that $A \wedge B$ is unsatisfiable, the interpolant I of A and B is a Boolean formula that is logically implied by A, unsatisfiable when conjoined with B, and contains only variables that are in A and B. Algorithms that compute Craig interpolants, which include recent improvements by Vizel [13], use the antecedent graph of the formula $A \wedge B$.

Optimizations

Reconstruction may result in a different antecedent graph than the one achieved during search. For example, assume that the antecedent graph in Fig 1 was produced by a SAT solver. During reconstruction, we might be able to produce a smaller antecedent graph that has fewer core clauses, core lemmas, and core arcs. An example of such an optimized antecedent graph is shown in Fig. 3.

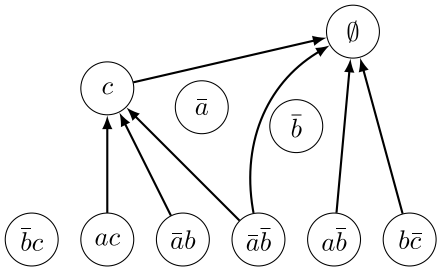

Fig. 3. Optimized antecedent graph for the example formula.

Aside from reconstruction of the antecedent graph, we also propose strategies to minimize the number of core clauses, core lemmas, or core arcs. We noticed that there is a trade-off between optimizing these different aspects (see Section VII-B).

Minimizing Core Clauses: The smaller the number of core clauses, the closer the one gets to a MUS. Hence, trying to minimize the set of core clauses during reconstruction could reduce the cost to extract a MUS.

Minimizing Core Lemmas: By reducing the number of lemmas in a clausal proof, checking costs are reduced, potentially enabling the use a mechanically-verified proof checker. Furthermore, a smaller number of lemmas can also increase MUS extraction by reducing the number of internal nodes in the antecedent graph.

Minimizing Core Arcs: The number of core arcs is related to the size of a resolution refutation of the core clauses. By minimizing the core arcs during the reconstruction, a smaller resolution refutation is obtained which can improve the speed of validating a resolution proof.

IV. Backward Reverse Unit Propagation

This section describes a method [1] to validate clausal proofs. The backward checking variant (see Section IV-B) can also be used to reconstruct an antecedent graph for a given proof. As far as we know, there is no implementation of this method available. We noticed that the method described in [1] can be very expensive when used on clausal proofs from state-of-the-art CDCL solvers. Sections V-A and VI discuss two optimizations that make this method much more efficient.

A. Forward Checking

Forward checking validates *each* lemma of a proof, in the order that they were learned, by checking if they have the RUP property. Forward checking is simple to implement [1], [3], [14], relatively easy to parallelize, and can start as soon as a clause is learned. However, this approach may check lemmas that are not required to validate a proof.

B. Backward Checking

Backward checking validates lemmas in the reverse order that they were learned. The advantage of checking a proof backward is that while validating a lemma, one can mark all the clauses that are used to determine that the lemma has the RUP property. When an unmarked lemma is encountered during a backward loop, the lemma is skipped. This can significantly reduce the checking costs by skipping lemmas during proof checking. Another advantage of backward checking is that it produces an unsatisfiable core from the original input clauses. This procedure can be used to *trim* formulas when computing minimal unsatisfiable cores.

Backward checking is more complex, however, because the checker needs to compute which clauses and lemmas were used for each lemma in a proof. Because of this complexity, it is harder to trust or verify a backward checking algorithm. Furthermore, the computation of the core clauses makes the procedure more costly. If only a few lemmas can be skipped, backward checking can be more costly than forward checking. Checking can only start when the solver has terminated, preventing an implementation that solves and checks in parallel. Finally, backward checking is hard to parallelize.

Fig. 4 shows the pseudo-code of the backward checking algorithm. Its input is the original formula F and a stack of learned lemmas S. The top of the stack is the last learned clause. For a refutation, this is typically the empty clause $\emptyset$. Initially, all clauses and lemmas are unmarked (line 1). A stack without the empty clause is not a refutation (line 2). The empty clause is marked (line 3). We pop lemmas from the stack until we find a marked lemma (lines 4-6), the first of which will be the empty clause. For marked lemmas, we validate that this lemma has the RUP property with respect to the original formula and all remaining lemmas in the stack (line 7). If the check succeeds, then the clauses and lemmas that were used during BCP are marked. The algorithm succeeds if it was able to validate all lemmas in S (line 9).

```
backwardRUP (CNF formula $F$, stack $S$ of lemmas)
1   forall $C \in F \cup S$ do core [$C$] = 0
2   if $\emptyset \notin S$ return "invalid refutation"
3   core [$\emptyset$] = 1
4   while $S$ is not empty do
5       $L := S.pop()$
6       if core [$L$] then
7           if BCP ($F \cup S, \bar{L}$) = "failed" then
8               return "failed"
9   return "refutation validated"
```

Fig. 4. Pseudo-code of the backward reverse unit propagation procedure.

Each lemma is validated by checking that a lemma L has the property RUP; this is performed by the BCP procedure (Fig. 5). BCP has two inputs: the set of clauses FS consisting of the original formula F and the remaining lemmas in the stack S, and the assignment τ that falsifies a lemma L (denoted by $\bar{L}$). During the procedure, a set of clauses U is maintained of all clauses that have become unit (line 1). The procedure terminates when the current assignment τ falsifies a clause in FS, by calling the *MarkCore* procedure (lines 3-4). Otherwise, it extends τ and updates U for each discovered unit clause (lines 5-7). If there are no more unit clauses and no clause has been falsified, the algorithm returns "failed" (line 8).

```
BCP (set of clauses $FS$, assignment $\tau$)
1   $U := \emptyset$
2   forever do
3       if $\exists\, C \in FS$ s.t. $\tau(C) = $ f then
4           return MarkCore ($U$, $C$)
5       if $\exists\, C \in FS$ s.t. unit($C$, $\tau$) then
6           $U.push(C)$
7           $\tau := \tau \cup$ unit$(C, \tau)$
8       else return "failed"
```

Fig. 5. Pseudo-code of the unit propagation (BCP) procedure.

The *MarkCore* procedure works as follows. The falsified clause R is marked (line 1). Then, any unit clauses that were found during BCP (in stack U) are examined in reverse order. If resolution is possible between the clause C on the top of the stack and resolvent R (which was the originally falsified clause), then C is marked and the resolvent R is updated by applying the resolution step $R := R \bowtie C$. When the stack is empty, the resulting R is falsified by the input assignment τ of the BCP procedure that called *MarkCore*.

```
MarkCore (unit stack $U$, clause $R$)
  core [$R$] := 1
  while $U$ is not empty do
    $C := U.pop()$
    if $C$ and $R$ have exactly one clashing literal then
      core [$C$] := 1
      $R := R \bowtie C$
  return "succeeded"
```

Fig. 6. The MarkCore procedure marks all clauses and lemmas that were involved in validating a lemma.

The *MarkCore* procedure is similar to the *analyzeFinal* [15] procedure that is used in CDCL solvers that support assumptions (decisions at level 0) also known as the last unique implication point.

Notice that the core arcs are not stored in this approach. This reduces the memory consumption significantly as compared to storing the antecedent graph during search. We can calculate how many core arcs would have been in the graph by summing up how often lines 1 and 5 of *MarkCore* are executed. Also, one can obtain the full antecedent graph by inserting edges from R (line 1) or C (line 5) to the current lemma L (in the backwardRUP procedure).

V. Adding Information to Clausal Proofs

The main disadvantage of clausal proof checking and trimming is the computational cost. Two methods have been proposed that add extra information in proofs to reduce the costs. The first method adds deletion information [14] (Section V-A) and the second method adds antecedents [6] (Section V-B).

A. Adding Deletion Information

The RUP checking algorithm presented in the prior section is costly for large proofs. The costs of verifying large proofs can be one-to-two orders of magnitude larger than the solving time. SAT solvers aggressively delete learned clauses during search whereas a RUP checking algorithm can only add lemmas.

In order to combat this disadvantage, we proposed to extend proof logging with clause deletion information [14][2]. In our proposed proof format, called DRUP (for delete reverse unit propagation), one can add lemmas to the formula (exactly in the same way as in the RUP format) and delete lemmas from the formula. Deleted lemmas have a d prefix. Fig. 7 shows an example CNF formula and a refutation for that formula in the DRUP format. The tool presented in this paper is the first proof checker that supports the DRUP format.

[2]The paper has been accepted with minor revisions and is available on EasyChair as a supplement.

CNF formula

```
p cnf 4 8
 1  2 -3  0
-1 -2  3  0
 2  3 -4  0
-2 -3  4  0
 1  3  4  0
-1 -3 -4  0
-1  2  4  0
 1 -2 -4  0
```

DRUP proof

```
    1  2  0
d   1  2 -3  0
    1  0
d   1  3  4  0
d   1 -2 -4  0
    2  0
d   2  3 -4  0
    0
```

Fig. 7. An example of a CNF problem in the typical DIMACS format (left) and a refutation for that formula in the DRUP format (right). Whitespaces can be of any length; spacing is used to improve readability. 0 marks the end of clauses and lemmas.

```
backwardDRUP (CNF formula $F$, stack $S$ of lemmas)
  forall $C \in F \cup S_A$ do core [$C$] = 0
  if $\emptyset \notin S_A$ return "invalid refutation"
  core [$\emptyset$] = 1
  while $S$ is not empty do
    $\langle L, flag \rangle := S.pop()$
    if $flag$ = "" and core [$L$] then
      if BCP $((F \cup S_A) \setminus S_D, \bar{L})$ = "failed" then
        return "failed"
  return "refutation validated"
```

Fig. 8. Pseudo-code of the backward DRUP procedure.

Given a stack of of labelled lemmas S, the set S_A denotes the lemmas in S with no label, while the set S_D denotes the lemmas in S with label d. In order to exploit deletion information, the top level procedure needs to be modified in three places. First, the $\emptyset$ should be in S_A (line 2). Second, we ignore other tests if the flag of a lemma is d (line 6). Third, all clauses in $F \cup S_A$ which are also in S_D are ignored during BCP.

B. Extending RUP with Antecedents

Another approach annotates RUP proofs with antecedent information [6]; we refer to these proofs as *extended RUP* proofs. Picosat [6] can emit extended RUP proofs. The reason why clausal proofs are expensive to validate is that the number of clauses that become unit during BCP (i.e., the set U) contains many clauses that were not required to show that the RUP property holds. In extended RUP proofs, each lemma in the proof is extended with a set of clauses that is sufficient to validate that lemma. Typically, this set contains the antecedents of the lemma. By restricting BCP to the set of clauses provided with each lemma (instead of all clauses of $F \cup S$), one can significantly decrease the time to validate a proof.

In order to emit an extended RUP proof, a SAT solver needs to store and maintain the antecedents of all lemmas; this requires a lot of memory and can significantly reduce the solving time. Furthermore, the number of antecedents can be an order of magnitude larger than the number of literals in a lemma. Hence, extended RUP proofs can be an order of magnitude larger than RUP proofs. In contrast, DRUP proofs are usually only twice as large as RUP proofs. Finally, extended RUP proofs must contain a list of the clauses in the input formula (since they will act as antecedents). This means that a checker also needs to validate that clauses that are claimed to be in the input formula are indeed present.

VI. Preferring Core Clauses during BCP

We observed that for many benchmarks, only a fraction of the original clauses and lemmas will be in the final core. To improve the speed of the checking algorithm, we considered an alternative implementation for BCP that prefers marked clauses and lemmas to unmarked clauses and lemmas. The pseudo-code of this algorithm is shown in Fig. 9.

```
CoreFirstBCP (set of clauses $FS$, assignment $\tau$)
  $U := \emptyset$
  forever do
    if ($\exists\, C \in FS$ s.t. $\tau(C) = \mathbf{f}$) and (core [$C$]) then
      return MarkCore ($U$, $C$)
    if ($\exists\, C \in FS$ s.t. unit($C$, $\tau$)) and (core [$C$]) then
      $U.push(C)$
      $\tau := \tau \cup$ unit($C, \tau$)
    else if $\exists\, C \in FS$ s.t. $\tau(C) = \mathbf{f}$ then
      return MarkCore ($U$, $C$)
    else if $\exists\, C \in FS$ s.t. unit($C$, $\tau$) then
      $U.push(C)$
      $\tau := \tau \cup$ unit($C, \tau$)
    else return "failed"
```

Fig. 9. BCP preferring clauses and lemmas which are in the core.

Ryvchin [16] proposed postponing unit propagation on *interesting constraints*, a subset of the clauses in the original formula. We observed that the number of core clauses and core lemmas becomes smaller when one postpones unit propagation on all clauses and lemmas that are not yet in the core.

VII. Experimental Evaluation

To demonstrate the usefulness of the DRUP-trim tool[3], we experimented with it on the application benchmarks from the SAT 2009 competition. We ran our tests on a system with a 4-core Intel Core i7 2.6GHz processor, 16GB of RAM, and 1TB of disk space running MacOS X 10.8.3. Throughout this section, we use two SAT solvers: Glucose 2.2 [17] and Picosat-953 [6]. Glucose is one of the fastest SAT solvers available and won the SAT 2012 Challenge. Our approach allows for Glucose preprocessing techniques [10] in the proof format, and therefore avoids the reconstruction problems presented by Belov [18]. Picosat is the fastest solver that can emit additional results such as resolution proofs, RUP proofs, and unsatisfiable cores.

This section describes two experiments using Picosat, Glucose, and our DRUP-trim tool. In the first experiment, we evaluate the time to emit proofs and the time to extract additional results. In the second, we evaluate the effectiveness of trimming as it relates to unsatisfiable cores, reduced clausal proofs, and reduced resolution proofs.

A. Comparing solving / checking / trimming times

For our first experiment, we determined how many unsatisfiable benchmarks of the SAT 2009 application suite could be solved by Glucose and Picosat. We ran Picosat with the option to emit extended RUP proofs. We modified Glucose to emit DRUP proofs (the input format for our DRUP-trim tool). This modification is about 40 lines of code, most of which are added to support preprocessing techniques.

Within a timeout of 900 seconds, Glucose with DRUP logging solved 123 instances, while Picosat with extended RUP proof logging solved only 81 instances. The benchmarks solved by Picosat were a subset of the benchmarks solved by Glucose. With an even larger timeout of 9000 seconds, Picosat was only able to solve 101 out of the 123 unsatisfiable benchmarks that Glucose can solve in 900 seconds. On most of the unsolved benchmarks, Picosat exhausted memory (limit 15 Gb). We noticed that turning on the proof logging in Picosat increased the memory consumption on some benchmarks by two orders of magnitude. In contrast, proof logging in our modified version of Glucose does not require additional memory because the proof is stored directly on disk. This experiment shows the disadvantage of producing additional results within a SAT solver: one is not able to produce a proof or core due to lack of memory on several benchmarks. We observed similar problems when using the state-of-the-art MUS extraction tool Muser [19] on the same instances.

Fig. 10 shows the runtime of Glucose and Picosat with proof logging enabled and the costs to validate the proofs emitted by Glucose. In our cactus plot, the data points for each line are sorted based the y-axis. The checking costs with four different settings of our DRUP-tool are also shown. Two settings use forward checking to validate all lemmas in the proof and do not mark clauses. One setting ignores the deletion information in the proofs (denoted by RUP-checking, similar to the approach in [3]), while DRUP-trim forward uses this information (similar to the approach in [14]). The other two settings use backward checking and hence mark the clauses and lemmas in the core. The fastest setting is the one that uses the core-first BCP technique. The core-first BCP is usually faster than conventional BCP. However, for some benchmarks with hundreds of thousands of variables, conventional BCP outperforms core-first BCP.

Notice that for up to 80 instances, the solving time by Glucose and the DRUP-trim checking times are comparable. For the next 25 instances, checking takes about twice as long as compared to solving. The checking time is only slower for a handful of instances. Yet, DRUP-trim with core-first BCP was

[3]DRUP-trim is available at http://cs.utexas.edu/~marijn/drup-trim/. If the paper is accepted, we will make the tool available as open source.

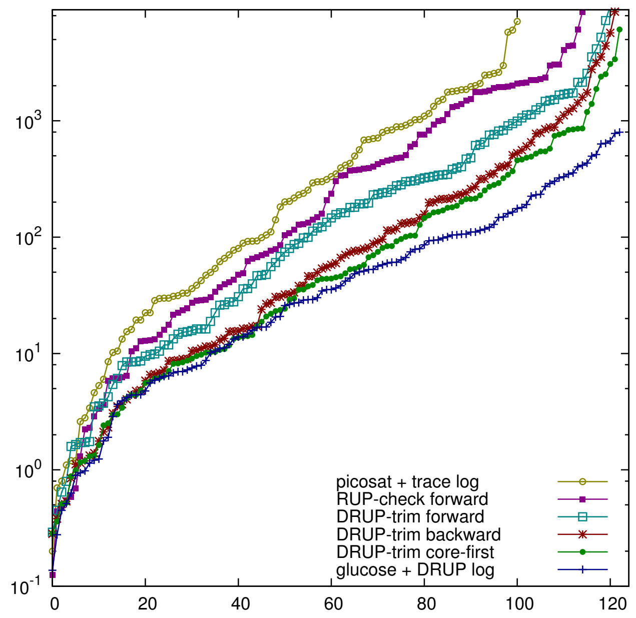

Fig. 10. Cactus plot of comparing the time to solve application benchmarks and the time to validate the emitted proofs. The plot shows the running time for the solvers Glucose with DRUP logging and Picosat with extended RUP logging. Additionally, the costs of checking the DRUP proofs with DRUP-trim are shown. Notice that the y-axis (time) uses a logarithmic scale.

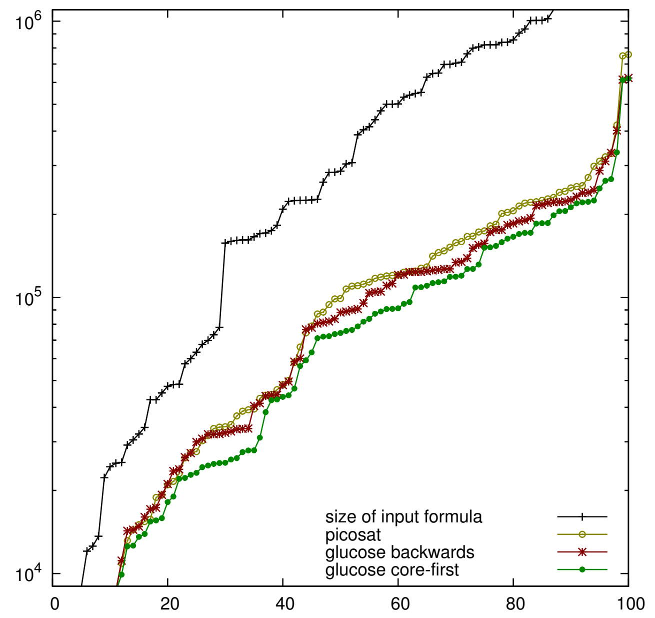

Fig. 11. Cactus plot of comparing the size (in the number of clauses) of the trimmed formulas. Notice that the y-axis (size) uses a logarithmic scale.

able to check all proofs within a timeout of 900 seconds. We believe it is now feasible to check all unsatisfiability results.

B. Comparing the Quality of Trimming

We restrict the experiments to the 101 benchmarks of the SAT 2009 application suite that Picosat with extended RUP logging was able to solve within the timeout of 9000 seconds and a memory limit of 15 Gb. Recall that Glucose could solve all these instances in less than 900 seconds.

Fig. 11 shows the size of the unsatisfiable cores produced by Picosat that stores the antecedent graph and by Glucose using our DRUP-trim tool. We experimented with two variants: one with conventional BCP and one with core-first BCP. In Fig. 10 we see that core-first BCP reduces the validation cost. Now we see that core-first BCP is more effective in trimming the formula (i.e., results in a smaller core). Furthermore, the cores produced by Picosat are larger than the ones produced by DRUP-trim using the Glucose proof. This suggests that it is not benefical to store the antecedent graph during search in order to trim a formula.

Our results use the DRUP-trim tool only once. We noticed that in most cases, the core can be further reduced by applying DRUP-trim multiple times. This can be done in two ways: use the reduced clausal proof or compute a new proof for the core clauses using a SAT solver. The best approach depends on the benchmark. For some benchmarks, a SAT solver produces a larger proof for the core clauses. In those cases, one should use the reduced proof. However, if a SAT solver is able to produce a smaller refutation for the core clauses, then those lemmas would be better for a next iteration.

We modified Picosat such that it can emit a DRUP proof without storing the antecedent graph (which requires only 10 lines of code). We compared Picosat when storing the antecedent graph to the modified version. In the latter case, we computed the antecedent graph with our DRUP-trim tool using the DRUP proof that was produced by the modified version of Picosat.

Fig. 12 shows the results. The alternative approach (i.e., using a DRUP proof) produces smaller unsatisfiable cores and a smaller clausal proofs (consisting of the core lemmas). Recall that the alternative approach does not store the core arcs and uses considerably less memory (1% to 10%).

The native core approach in Picosat produces fewer core arcs compared to the approach using DRUP-trim with the core-first BCP strategy. This is not surprising: by postponing BCP on clauses and lemmas that are not in the core, several arcs to core clauses and core lemmas are produced. A possible way to reduce the number of core arcs is to run the DRUP-trim tool twice. First, using core-first BCP using the original formula and proof; and second, using conventional BCP using the core clauses and the core lemmas.

VIII. Conclusion

We presented the tool DRUP-trim that can efficiently check clausal proofs and can produce additional results including unsatisfiable cores and reduced clausal proofs. DRUP-trim is the fastest clausal proof checker available. This tool also makes it more convenient to check clausal proofs and to obtain additional results. A slightly modified version of DRUP-trim will be used to check the unsatisfiability results of the upcoming SAT Competition 2013.

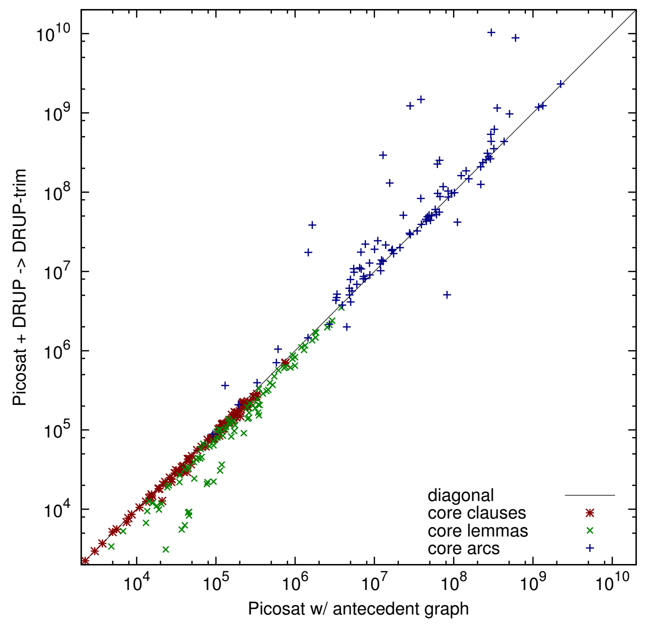

Fig. 12. Comparing the number of core clauses, lemmas, and arcs of Picosat with antecedent graph support and a modified version emitting a DRUP proof.

DRUP-trim was able to verify all unsatisfiability claims of the state-of-the-art SAT solver Glucose 2.2 on the benchmarks of the SAT 2009 application suite. Other work on checking of unsatisfiability results [1], [2], [3], [4], [11], [16] only show results for selected (small) sets of benchmarks. We compared our DRUP approach with Picosat, the strongest solver that can emit resolution proofs. In contrast to emitting a DRUP proof, building a resolution proof during search can significantly increase the memory requirements of the solver. As a consequence Picosat resulted in memory outs on about 20 application benchmarks that Glucose 2.2 was able to solve in 900 seconds. This suggests that resolution proofs are not a viable method for checking the unsatisfiability results in a general setting, such as the SAT Competition.

The best tools for extracting minimal unsatisfiable cores, such as Muser [19], and computing interpolants, such as CNF-ITP [13], are based on resolution proofs. But there are two important drawbacks. First, similar to proof checking, some benchmarks cannot be solved when one builds a resolution proof during search. Second, none of the top-tier solvers supports the emission of a resolution refutation. Current approaches [19], [11], [16], [13] rely on either Picosat or Minisat [20] which are no longer the strongest solvers. We propose to use DRUP proofs as input for tools that compute MUSes and interpolants. This makes it easy to use any SAT solver for those tools.

One of our optimizations, the core-first BCP technique, facilitates the computation of smaller unsatisfiable cores and smaller reduced clausal proofs when compared to resolution-style methods. However, the number of core arcs increases when this technique is used. One focus of our future work will be to find a technique for DRUP-trim that makes it possible to reduce the number of core arcs as well.

Acknowledgment

The authors are supported by DARPA contract number N66001-10-2-4087.

References

[1] E. I. Goldberg and Y. Novikov, “Verification of proofs of unsatisfiability for CNF formulas,” in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*. IEEE, 2003, pp. 10 886–10 891.

[2] L. Zhang and S. Malik, “Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications,” in *DATE*, 2003, pp. 10 880–10 885.

[3] A. Van Gelder, “Verifying RUP proofs of propositional unsatisfiability,” in *International Symposium on Artificial Intelligence and Mathematics (ISAIM)*. Springer, 2008.

[4] A. Darbari, B. Fischer, and J. Marques-Silva, “Industrial-strength certified SAT solving through verified SAT proof checking,” in *International Colloquium on Theoretical Aspects of Computing (ICTAC)*. Springer-Verlag, Sep. 2010, pp. 260–274.

[5] R. Brummayer, F. Lonsing, and A. Biere, “Automated testing and debugging of SAT and QBF solvers,” in *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing*, ser. SAT’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 44–57. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14186-7_6

[6] A. Biere, “PicoSAT essentials,” *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, vol. 4, no. 2-4, pp. 75–97, 2008.

[7] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell, “Efficient generation of unsatisfiability proofs and cores in SAT,” in *LPAR-17*, ser. LNCS, I. Cervesato, H. Veith, and A. Voronkov, Eds., vol. 5330, Springer. Springer, 2008, pp. 16–30.

[8] M. Heule, M. Järvisalo, and A. Biere, “Efficient cnf simplification based on binary implication graphs,” in *SAT*, ser. Lecture Notes in Computer Science, K. A. Sakallah and L. Simon, Eds., vol. 6695. Springer, 2011, pp. 201–215.

[9] M. J. H. Heule and H. van Maaren, *Look-Ahead Based SAT Solvers*. Handbook of Satisfiability, IOS Press, February 2009, ch. 5, pp. 155–184.

[10] N. Eén and A. Biere, “Effective preprocessing in SAT through variable and clause elimination,” in *Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2005, pp. 61–75.

[11] A. Nadel, “Boosting minimal unsatisfiable core extraction,” in *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2010, pp. 221–229.

[12] K. L. McMillan, “Interpolation and SAT-based model checking,” in *Computer Aided Verification (CAV)*. Springer, 2003, pp. 1–13.

[13] Y. Vizel, V. Ryvchin, and A. Nadel, “Efficient generation of small interpolants in CNF,” in *Computer Aided Verification (CAV)*. Springer, 2013, p. to appear.

[14] M. J. H. Heule, W. A. Hunt, Jr., and N. Wetzler, “Bridging the gap between easy generation and efficient verification of unsatisfiability proofs,” *Software Testing, Verification, and Reliability (STVR): Special Issue on Tests and Proofs*, 2013, accepted with minor revisions.

[15] N. Eén, A. Mishchenko, and N. Amla, “A single-instance incremental sat formulation of proof- and counterexample-based abstraction,” *CoRR*, vol. abs/1008.2021, 2010.

[16] V. Ryvchin and O. Strichman, “Faster extraction of high-level minimal unsatisfiable cores,” in *Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2011, pp. 174–187.

[17] G. Audemard and L. Simon, “Predicting learnt clauses quality in modern SAT solvers,” in *International Joint Conference on Artifical Intelligence (IJCAI)*, C. Boutilier, Ed., 2009, pp. 399–404.

[18] A. Belov, M. Järvisalo, and J. Marques-Silva, “Formula preprocessing in MUS extraction,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2013, pp. 108–123.

[19] A. Belov and J. Marques-Silva, “Accelerating MUS extraction with recursive model rotation,” in *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2011, pp. 37–40.

[20] N. Eén and N. Sörensson, “An extensible sat-solver,” in *SAT*, ser. LNCS, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518.