Linear-time Reductions of Resolution Proofs (full version)

Omer Bar-Ilan[1] Oded Fuhrmann[1] Shlomo Hoory[1] Ohad Shacham[1]
Ofer Strichman[2]

[1] IBM Haifa Research Laboratory. barilan|Odedf|shlomoh|ohads@il.ibm.com
[2] Information Systems Engineering, IE, Technion, Haifa, Israel.
ofers@ie.technion.ac.il

Abstract. DPLL-based SAT solvers progress by implicitly applying binary resolution. The resolution proofs that they generate are used, after the SAT solver's run has terminated, for various purposes. Most notable uses in formal verification are: extracting an *unsatisfiable core*, extracting an *interpolant*, and detecting clauses that can be reused in an *incremental satisfiability* setting (the latter uses the proof only implicitly, during the run of the SAT solver). Making the resolution proof smaller can benefit all of these goals: it can lead to smaller cores, smaller interpolants, and smaller clauses that are propagated to the next SAT instance in an incremental setting. We suggest two methods that are linear in the size of the proof for doing so. Our first technique, called RECYCLE-UNITS, uses each learned constant (unit clause) (x) for simplifying resolution steps in which x was the pivot, prior to when it was learned. Our second technique, called RECYCLE-PIVOTS, simplifies proofs in which there are several nodes in the resolution graph, one of which dominates the others, that correspond to the same pivot. Our experiments with industrial instances show that these simplifications reduce the core by $\approx 5\%$ and the proof by $\approx 13\%$. It reduces the core less than competing methods such as RUN-TILL-FIX, but whereas our algorithms are linear in the size of the proof, the latter and other competing techniques are all exponential as they are based on SAT runs. If we consider the size of the proof graph as being polynomial in the number of variables (it is not necessarily the case in general), this gives our method an exponential time reduction comparing to existing tools for small core extraction. Our experiments show that this result is evident in practice more so for the second method: rarely it takes more than a few seconds, even when competing tools time out, and hence it can be used as a cheap proof post-processing procedure.

1 Introduction

DPLL-based SAT solving became in the last few years the single most-used back-end engine for model checking and satisfiability modulo theories. While SAT solvers exist from at least the 1960's, and learning through derivation of conflict clauses exists from at least the 1980's (while recognized as implicitly derived by resolution), only in 2003 the question of how to produce a resolution proof

from a run of a DPLL solver was addressed in practice [23] and implemented. Most modern solvers nowadays are capable of producing such proofs. Further, there are now decision heuristics that are based on an understanding of the DPLL process as a resolution-based proof engine rather than as a search engine. Examples are Ryan's thesis [18] and his SAT solver SIEGE, which bias conflict-clause generation towards those that will more likely lead to other resolutions, and the work by Gershman et al. [7] on a model for explaining and designing decision heuristics based on an understanding of the SAT solving process as a resolution engine.

The resolution proofs that modern SAT solvers generate are used in a broad range of applications in formal verification and elsewhere. Prominent examples are:

- The resolution proof can be used for extracting an *unsatisfiable core*, which can then be used, e.g., in a proof-based abstraction-refinement procedure, as shown by Amla and McMillan [1, 2]. The unsatisfiable core was also used in the past for detecting the reasons for unsatisfiability in an underapproximation/refinement process for model-checking [9]. In the context of satisfiability modulo theories (SMT): the core is used for finding small *explanations* (see, for example, the recent work by Cimatti et al. [4], who suggest to invoke a tool for minimizing resolution proofs for this purpose) and in theory-specific decision procedures, such as the abstraction-based procedures for Presburger and bitvector formulas proposed by Kroening et al. and Bryant et al., respectively [11, 3].
- The resolution proof can be used for extracting an *interpolant* as part of a complete model-checking technique, as suggested by McMillan in [14].
- The proof can be used for detecting clauses that can be reused in an *incremental satisfiability* setting [19, 21], such as the one used in Bounded Model-Checking. The analysis of the proof is done implicitly, during the run of the SAT solver (in contrast to the first two uses), and is required in order to check whether all the clauses that were used for resolving a particular conflict clause are still expected to be present in the new SAT instance. If yes, this conflict clause can be reused.

Making the proof smaller by removing some of the nodes and removing literals from other nodes can benefit all of these goals. In all of these applications, however, reducing the size of the core/interpolant/reused-clauses has an influence on the overall run time which is somewhat unpredictable and only vaguely known. Further, the proof reduction component is called many times during the overall solving process. As a result, best overall results are most likely to be achieved with a limited investment in such reductions.

The script RUN-TILL-FIX by Zhang and Malik [23] extracts a core and attempts to minimize it by simply running the SAT solver on it repeatedly until reaching a fix-point. Achieving fast reductions with this tool is not always possible, as it requires a normal SAT run. The goal of achieving fast reductions was first addressed in the work by Gershman et al. [6], based on analyzing the proof graph and trying to restructure it with the aid of a SAT solver. The tasks the

SAT solver is asked to solve by their script TRIM-TILL-FIX are closely related, and hence incremental satisfiability makes this process relatively fast. There are also many published techniques for finding *minimal* cores, i.e., an unsatisfiable subset of clauses from which no clause can be removed without making it satisfiable (this is also known in the literature by the name MUS, for Minimal Unsatisfiable Subformula). Some works in this direction from the last four years are [13, 16, 10, 15, 5, 8], all of which are worst-case exponential. The complexity of the decision problem corresponding to finding the minimal unsatisfiable core is D^P-complete[1] [17]. A minimal core (in contrast to the *minimum* core) is not unique and depends on the starting point, like all the methods we are aware of (including the current one). Therefore whether the core found is minimal or just 'small', as indicated in [6], has little significance in practice.

The proof reductions techniques we suggest here are linear in the size of the proof graph. The proof graph itself can be exponential in the number of variables, and hence in the worst case our procedure is still exponential in the number of variables, like RUN-TILL-FIX. However, if we assume that in practice the size of the proof graph is not more than the number of variables multiplied by some polynomial with a bounded exponent, there is a gap of an exponent between our method and the competing tools, as they can still be exponential in the number of variables regardless of the size of the proof. Our techniques typically reduce the core significantly less than the exponential methods, but do so much faster. They also reduce the size of the proof itself whereas a procedure like RUN-TILL-FIX increases it. We therefore consider them as useful tools in the context of short time-outs and where the size of the proof matters.

The first method we suggest is called RECYCLE-UNITS. The idea is to remove edges from the proof graph by using information that is inferred by the SAT solver only *after* the resolutions at the nodes adjacent to these branches were made originally. For example, if (x) is a unit clause that was learned by the SAT solver, it can be used for simplifying resolution inferences that used x as the pivot prior to learning this clause. If not carefully applied, however, this may lead to circular reasoning.

The second method is called RECYCLE-PIVOTS. It is based on the following observation. For simplicity assume that the proof graph is a tree. Let n_1 and n_2 be two nodes on the same path in the proof tree such that n_1 is closer to the root. Further, assume that the pivot variable associated with both nodes is the same. Our convention is that proofs progress from top to bottom, from the premises (also called the *axioms*) of the proof to its consequent. We also follow the convention by which the right parent of each node contains the negative phase of the pivot variable, and the left parent contains the positive phase of this variable. Assume that n_2 is on the right branch of n_1. In this case, we will show, the left incoming branch of n_2 can be pruned, and the proof rewritten without it in a way that the resulting proof is a legal resolution proof with a

[1] D^P is the class containing all languages that can be considered as the difference between two languages in NP, or equivalently, the intersection of a language in NP with a language in co-NP.

smaller core. Since in practice proof graphs are DAGs and not just trees, the procedure is somewhat more complicated as we later describe.

The two techniques tighten the proof, which means that the resulting proof uses a subset of the core used by the original proof, and there is an injective mapping between the target and source nodes, such that each target node is either equal or subsumes the source node to which it is mapped. As a theoretical curiosity, we note that the resolution proofs generated by our techniques cannot necessarily be generated by any modern SAT solver and hence by any core reduction technique that is based on rerunning such a solver. More details about this issue can be found in the appendix.

The rest of the paper is structured as follows: Section 2 summarizes some preliminaries necessary for the description of the technique (we assume, however, that the reader is familiar with SAT basics). Section 3 and 4 describe the two techniques by giving pseudo code and intuitive explanation of their correctness. Section 5 is dedicated to a formal proof of their correctness. We conclude in Sect. 6 with a description of the experiments we conducted and their results.

2 Preliminaries

A literal is an instance of a Boolean variable or its negation. A pair of literals corresponding to a variable and its negation are called *complementary*. A clause is a (possibly empty) disjunction of literals, and a CNF formula is a conjunction of clauses.

2.1 Inference by Resolution

The process in which DPLL SAT solvers infer new conflict clauses can be interpreted as applying the *binary resolution* inference rule:

$$\frac{(l \vee l_1 \vee .. \vee l_n) \qquad (\neg l \vee l'_1 \vee .. \vee l'_n)}{(l_1 \vee .. \vee l_n \vee l'_1 \vee .. \vee l'_n)} \ (\textsc{Resolution}). \tag{1}$$

The variable l is called the *pivot variable* (also called 'resolution variable' in the literature).

Let *Res* be a function that receives two clauses with complementary literals as input, and returns the consequent of the resolution rule applied to these two clauses, as output. More formally:

Given clauses $C_1 = (l \vee l_1 \vee \ .. \ \vee l_n)$ and $C_2 = (\neg l \vee l'_1 \vee l'_2 \vee \ .. \ \vee l'_n)$,

$$Res(C_1, C_2) = (l_1 \vee \ .. \ \vee l_n \vee l'_1 \vee \ .. \ \vee l'_n) \ .$$

Resolution is known to be a sound and complete proof system for CNF formulas. Specifically, a CNF formula φ is unsatisfiable if and only if there exists a resolution proof of the empty clause using φ's clauses as premises.

Definition 1 (Resolution Graph). *A resolution graph corresponding to a resolution proof is a directed acyclic graph (DAG), where the nodes represent clauses and for every pair of nodes C_i, C_j, (C_i, C_j) is an edge if and only if C_i is an antecedent of C_j in the resolution proof.*[2]

Example of a resolution graph can be seen in Fig. 1[a]. Every resolution proof can be represented by a resolution graph. If a resolution graph has a single sink, it is called the *consequent* of the proof. The root nodes are the premises of the proof. For a given node, the root nodes that can reach it on the proof graph are called its *core*. Specifically, if the consequent is an empty clause then its core is called the *unsatisfiable core*.

The resolution described so far is known by the name *general resolution*. Two well-known restrictions of general resolutions that we will mention later on are *tree-like resolution*, which means that the proof graph corresponding to the proof is a tree rather than a DAG, and Tseitin's *regular resolution* [20], which means that along each path no variable is used twice as a pivot. Both of these restrictions may cost in a penalty of an exponent in the size of the proof. In other words, there are formulas that can be proven with general resolution in a polynomial number of steps, but only with an exponential number of tree-like resolution or regular resolution.

How do resolution proofs relate to proofs of SAT solvers? Modern DPLL SAT solvers generate *conflict clauses* [22] during their run, which are implicitly inferred from other clauses by a chain of (general) resolutions. Hence, a proof of unsatisfiability given by solvers have original clauses as their roots, and conflict clauses as their internal nodes. We refer the reader to [23] for more details.

We are going to use a variant of the resolution graph in which a single parent is allowed, if this parent is associated with the same clause as the child. The resolution proof corresponding to this graph is derived by first eliminating all nodes with a single parent (removing such a node n and connecting n's single parent to n's children), and continuing as before. This extension is convenient for simplifying the algorithm and later on the proofs, but is not essential.

3 Recycling Learned Unit Clauses

Some of the conflict clauses learned during the run of a SAT solver are unit clauses, e.g., (x). In such a case we say that the SAT solver inferred the *constant value* of the variable constrained by this clause ($x = true$ in this case). These constants can be used to rewrite the proof starting from those parts of the proof that were inferred prior to learning these constants, an action that reduces the overall size of the proof and its core. In other words, the SAT solver can only apply resolution to clauses in its clause database (which, recall, is continuously

[2] Note that we use the convention by which the edges are in the direction of the proof, i.e., from premises to consequent. For practical purposes it is common to build this graph with edges (pointers) pointing in the other direction, because this facilitates a search for the core.

updated with new conflict clauses). Further, it can only use resolution variables which at that point in time are unassigned. If at a later stage of the computation the same resolution variable is proved to have a constant value then the proof can be regenerated taking this information into account.

Algorithm 1 presents RECYCLE-UNITS, which is the first of a two-step algorithm for recycling unit clauses. The second step is RECONSTRUCT-PROOF, which is presented in Algorithm 2. The two algorithms use the following notation. P is a resolution proof of the empty clause, and $P.sink$ is the node representing the empty clause. For a given node n in P, $n.C$ is the clause represented by n, $n.L$ and $n.R$ are the left and right parents of n respectively, and $n.piv$ is the pivot variable used to resolve $n.C$ from $n.L.C$ and $n.R.C$. Recall that we use the convention by which the left parent includes the positive phase of the pivot, and the right parent includes its negative phase. If l is a literal, we denote by $var(l)$ its corresponding variable. When $n.C$ is a unit clause, we sometimes refer to it as a literal rather than a clause, when the meaning is clear from the context. For example $var(n.C)$ is the variable corresponding to the literal in the unit clause $n.C$.

Algorithm 1: RECYCLE-UNITS(PROOF P)

1: Let U be the set of nodes representing constants proved in P;
2: **for** each $u \in U$ **do**
3: Mark the (recursive) antecedents of u;
4: **for** each unmarked $n \in P$ **do**
5: **if** $n.piv == u.C$ **then**
6: $n.L = u$;
7: **else if** $n.piv == -u.C$ **then**
8: $n.R = u$;

RECYCLE-UNITS iterates over all constants that were proved in P. Let u be a node representing such a constant, and assume for now that this constant is a positive literal (i.e., $u.C$ is a positive literal). First, in line 3, RECYCLE-UNITS marks the nodes in its antecedents closure, i.e., the nodes that can reach U in P. Then, it searches unmarked nodes for those that represent a resolution step using $var(u.C)$ as pivot. Let n be such a node. According to our convention $u.C \in n.L.C$ and $\neg u.C \in n.R.C$. In line 6 RECYCLE-UNITS replaces the left parent of n from $n.L$ to u, i.e., it disconnects the edge $(n.L, n)$ and connects instead (u, n). If $u.C$ is a negative literal, then the edge is shifted from $(n.R, n)$ to (u, n) in line 8.

At this stage the proof is no longer a legal resolution proof, and hence a reconstruction phase begins by calling RECONSTRUCT-PROOF, which appears in Alg. 2.

Example 1. Consider the partial proof graph that is depicted in Fig. 1a. The unit clause C8 is proven only after the resolution of C3, which uses the variable ‘1’ (the unit of C8) as its pivot. Recycle-Units begins by marking clauses that were used for proving C8 – clauses C5 and C6 in this case. It then identifies C3 as an unmarked node that uses ‘1’ as pivot and rewires the proof – in this case disconnects (C2,C3) and adds instead the edge (C8,C3), as can be seen in Fig. 1b. It is left to reconstruct the proof so it becomes a legitimate resolution proof once again.

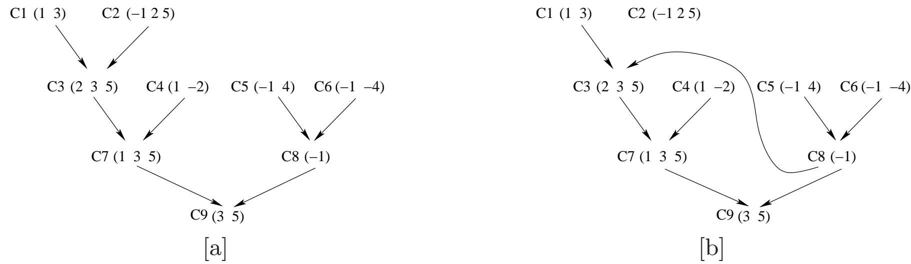

Fig. 1. [a] Part of a resolution proof [b] After Recycle-Units and before Reconstruct-Proof.

□

Implementation of Recycle-Units The most time-consuming component of Algorithm 1 is Step 3. Recall that the purpose of this step is to prevent cycles after we connect a unit clause to another node. Rather than traversing the graph backwards each time (which would make this method quadratic in the size of the graph), our implementation maintains at each unit node pointers to all immediate descendent units. Denote by G_U the resulting graph of units, i.e., the nodes of G_U are the units in the resolve graph and the edges are defined by the list of pointers that we maintain with each such node. In Step 3, when considering connecting a unit clause u to another node c, our tool temporarily makes this connection and checks if it can reach itself on G_U. If the answer is yes, we undo this temporary connection and continue. Otherwise we keep the connection and update the list of pointers in the units that are immediate antecedents of u so they now also point to u. This method makes the solution quadratic in the number of units. We can always bound the number of units that we consider (regardless – in practice it is small comparing to the size of the proof) and hence refer to this element in the complexity analysis as a constant.

Let us now shift the focus to Reconstruct-Proof. The proof graph given to Reconstruct-Proof as input has a subset of the nodes of the original proof graph. This is because only edges are shifted up to this point, and these shifts may disconnect parts of the graph (e.g., in Example 1, node C2 has been

disconnected). Note that only the graph connected to the sink node is sent to Reconstruct-Proof.

Reconstruct-Proof, appearing in Alg. 2, is a procedure that, given such a "broken" proof P and a node n (initially the empty clause), reconstructs a legal resolution proof of n (we will define formally the nature of this broken proof in Sect. 5). The procedure is recursive starting from the node n. The base of the recursion are the root nodes. When n is not a leaf, there are several cases. If the pivot $n.piv$ is still present in its parents then $n.C$ is a resolution between them (see lines 10 and 11). If it is only present in one of them, say the positive phase is present in $n.L.C$, then the other node ($n.R$) replaces n. This is because we know that $n.R.C$ subsumes $n.C$. If it is contained in neither of its parents, then both $n.L.C$ and $n.R.C$ subsume $n.C$ and hence either one of them can replace $n.C$.

Algorithm 2: Reconstruct-Proof ("Broken" proof P, node n)

1: **if** n visited **then return**
2: mark n as visited;
3: **if** n is a root **then return**
4: **if** n has a single parent $n.L$ **then**
5: Reconstruct-Proof (P,$n.L$);
6: $n.C = n.L.C$;
7: **else**
8: Reconstruct-Proof (P,$n.L$);
9: Reconstruct-Proof (P,$n.R$);
10: **if** $n.piv \in n.L.C$ and $\neg n.piv \in n.R.C$ **then**
11: $n.C = Res(n.L.C, n.R.C)$;
12: **else if** $n.piv \in n.L.C$ and $\neg n.piv \notin n.R.C$ **then**
13: $n.C = n.R.C$;
14: $n.L = \text{nil}$
15: **else if** $n.piv \notin n.L.C$ and $\neg n.piv \in n.R.C$ **then**
16: $n.C = n.L.C$;
17: $n.R = \text{nil}$;
18: **else**
19: $side$ = one of $\{L, R\}$; $otherside$ = other side; ▷ Choose heuristically
20: $n.C = n.side.C$;
21: $n.otherside = \text{nil}$;

Example 2. The graphs in Fig. 2 show the steps of reconstruction for the proof graph in Fig. 1b.

□

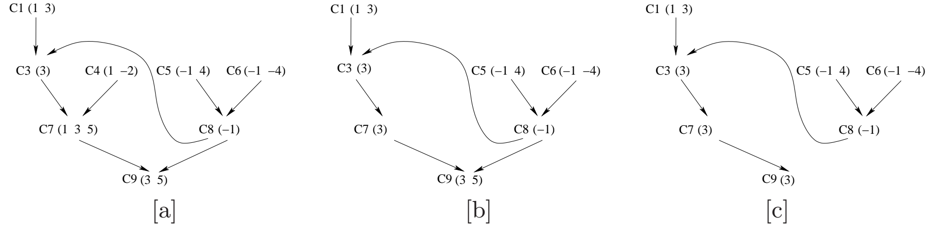

Fig. 2. The recursive steps, from left to right, of reconstructing the proof graph in Fig. 1b. Recall that single parent nodes are consistent with our definition of resolution graphs.

4 Recycling Pivots

The second technique we present is linear in the size of the proof and is also based on two steps, as in the previous case: in the first step we remove edges from the proof, and in the second step we reconstruct the proof using the same algorithm Reconstruct-Proof. The algorithm is based on the observation that along each path from root to sink, there is no need for resolving on the same variable more than once. If there is such a situation, then the redundant resolution steps can be avoided and consequently some of the branches of the proof can be pruned away. The correctness of this algorithm will be proven in Sect. 5. We note that this does not result in a regular resolution proof as defined in Sect. 2, because we apply it only to some parts of the graph as described below. If the resolution graph we start with happens to be a tree-like resolution, then the result of applying this step is indeed a regular resolution proof.

For simplicity of the presentation, assume for now that the proof is a tree rather than a DAG. Consider a node n with a pivot $n.piv$. We propagate $n.piv$ up the right branch (recall that according to our convention $n.R.C$ contains $\neg n.piv$) in a set called *Removable-Literals*, or RL for short. Similarly, we propagate $\neg n.piv$ up the left branch. Consider the right branch: if along this branch there is another node n' such that $n'.piv = n.piv$ then we can replace n' with $n'.R$. This means that the branch starting at $n'.L$ is pruned. The correctness of this operation is tied to the second step, which is the proof reconstruction in Reconstruct-Proof. Note that $n'.R.C$ is contained in $n'.C$ other than $\neg n.piv$. Reconstruct-Proof will effectively propagate $\neg n.piv$ down the branch until inevitably reaching n (a result of our assumption that this is a tree). At node n, $\neg n.piv$ will disappear again due to the resolution on $n.piv$ at n. As a result $n.C$ will subsume its original version.

In practice the input proof can be a DAG. This may cause a situation in which our node n' has paths to the sink not through n. This, in turn, may cause a situation that Reconstruct-Proof propagates $\neg n.piv$ all the way down to the sink, which contradicts our goal of producing a proof with an equal or stronger consequent. Another possible problem is that n' has paths to the sink node through both incoming edges of n, which nullifies our suggested technique.

There are two possible solutions to this problem. One, which is the solution taken in the pseudo-code described here (see line 4) and also in our implementation, is to propagate RL up (towards the roots) only as long as it is a tree. A more complicated solution, which we leave for future work, is to check whether all paths from n' to the sink go through the edge $(n.R,\ n)$. This can be done by computing *dominance* relation in the graph, for example with the Lengauer-Tarjan algorithm[12] (which runs in $O(|E| \log |V|)$ time). It might burden the computation, however, since the dominance relation has to be recomputed after each removal of an edge.

Algorithm 3: RECYCLE-PIVOTS(n, RL)

1: **if** n visited **then return**
2: Mark n as visited
3: **if** leaf **then** return
4: **if** n has more than one child **then** RL $= \{\}$
5: **if** $piv \notin$ RL and $\neg piv \notin$ RL **then**
6:     RECYCLE-PIVOTS($n.L$, RL $\cup \{\neg piv\}$)
7:     RECYCLE-PIVOTS($n.R$, RL $\cup \{piv\}$)
8: **else if** $piv \in$ RL **then**   ▷ this implies $\neg piv \notin$ RL
9:     $n.L =$ nil;
10:     RECYCLE-PIVOTS($n.R$, RL);
11: **else**   ▷ $piv \notin$ RL and $\neg piv \in$ RL
12:     $n.R =$ nil;
13:     RECYCLE-PIVOTS($n.L$, RL);

Example 3. Assume Fig. 3a represents the input proof for RECYCLE-PIVOTS. RECYCLE-PIVOTS propagates up the removable literals (denoted by *RL* in the drawing) and, owing to the fact that '2' is the pivot of C3 and that '2' is in RL, erases the edge (C1,C3). The proof after calling RECONSTRUCT-PROOF is depicted in Fig. 3b.

□

5 Proofs

Our goal is to prove the following two theorems.

Theorem 1. *Let P be a resolution unsatisfiability proof on the axioms A. Let P' be the result obtained by running* RECYCLE-UNITS *and then* RECONSTRUCT-PROOF *on P. Then P' is a valid resolution unsatisfiability proof on the axioms A', where $A' \subseteq A$.*

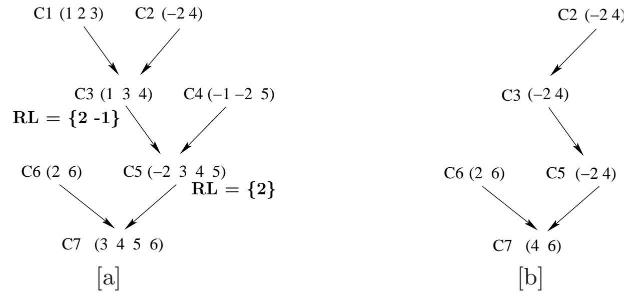

Fig. 3. For the input proof presented in drawing a, RECYCLE-PIVOTS erases the edge (C1,C3) and then calls RECONSTRUCT-PROOF, which results in the graph in drawing b. **RL** represents the Removable-Literals sets.

Theorem 2. *Let P be a resolution unsatisfiability proof on the axioms A. Let P' be the result obtained by running* RECYCLE-PIVOTS *and then* RECONSTRUCT-PROOF *on P. Then, P' is a valid resolution unsatisfiability proof on the axioms A', where $A' \subseteq A$.*

The proof of correctness relies on a notion of *e-resolution*, which we soon define. As we will prove, the graphs produced by RECYCLE-UNITS and RECYCLE-PIVOTS are e-resolution proof graphs, and RECONSTRUCT-PROOF transforms them back to resolution graphs.

It is convenient to represent resolution proofs as a DAG in which only the roots are labeled with clauses. The clauses labeling the other nodes, including the consequents, can be inferred from the topology of the graph and the roots, and hence are not considered as part of the representation. As an example, the right drawing of Fig. 4 represents the resolution graph corresponding to the proof graph on its left, which is based on our convention. Recall that single parents are allowed if the parent is labeled with the same clause as the child.

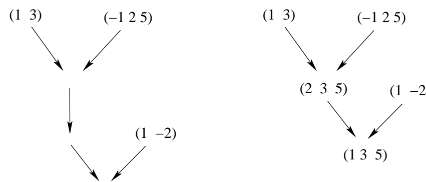

Fig. 4. The two graphs represent equivalent resolution proofs. The convention we use in this section corresponds to the left figure.

e-resolution is defined based on this convention, with the difference that each internal node n with two parents is labeled with a variable $n.piv$,[3] which in our case is the pivot used to infer $n.C$ at that node in the original proof. For nodes with two parents, instead of Eq. (1), e-resolution uses a more relaxed inference rule, of which (1) is a special case. Using the convention that $n.\text{piv}$ may occur only in $n.L.C$, and $\neg n.\text{piv}$ may occur only in $n.R.C$, the e-resolution inference rule is:

$$\begin{aligned} n.C &= \text{e-resolution}_{n.\text{piv}}(n.L.C, n.R.C) \\ &= (n.L.C \setminus \{n.piv\}) \cup (n.R.C \setminus \{\neg n.piv\}) \ . \end{aligned} \tag{2}$$

For example, for a node n labeled with a pivot x, and parents $n.L.C = (y_1 \vee y_2)$ and $n.R.C = (\neg x \vee y_3)$ we have $n.C = (y_1 \vee y_2 \vee y_3)$.

5.1 The Reconstruct-Proof Algorithm

In the following we denote by $\leq$ the subsumption relation, e.g., $C_1 \leq C_2$ means that C_1 has a subset of the literals of C_2.

Lemma 1. *Let P be an e-resolution proof of C. Then $P' = \text{Reconstruct-Proof}(P)$ is a resolution proof of C', where $C' \leq C$ and $A(P') \subseteq A(P)$.*

Proof. The claim follows by induction on the height of the proof (the length of the longest path from a root to the sink). The proof is a straight-forward analysis of the four possible cases for a node n as described by the table in Fig. 5. Since by the induction hypothesis we can assume that the proofs of $n.L.C$ and $n.R.C$ are legitimate resolution proofs, then clearly the second and third cases, which simply copy a parent node, result in a resolution proof of $n.C$. In the fourth case the choice is made heuristically to achieve best pruning, and the induction step applies to both choices. It is also clear that in all cases, at each step the clause labeling n can only be smaller than the one required by applying the e-resolution inference rule (2) to the updated $n.L.C$ and $n.R.C$. This of course applies to the consequent clause as well.

$n.C$	$n.\text{piv} \in n.L.C$	$\neg n.\text{piv} \in n.R.C$
$\text{Res}(n.L.C, n.R.C)$	yes	yes
$n.L.C$	no	yes
$n.R.C$	yes	no
$n.R.C$ or $n.L.C$	no	no

Fig. 5. The four cases discussed in the proof of Lemma 1.

[3] This is in contrast to a standard resolution graph, in which the pivots can be inferred from the topology and axioms.

5.2 The Recycle-Units Algorithm

Recycle-Units is a special case of a more general procedure, which we call **subsumption**. It generalizes Recycle-Units in the sense that it does not only recycle unit clauses, rather any learnt clause that subsumes other clauses in the proof. Our proof will refer to subsumption, and the correctness of Recycle-Units is then implied.

Consider two nodes p, m in the proof P such that $p.C \leq m.s.C$ for some side $s \in \{L, R\}$. Then, as long as no cycle is produced, we can set $m.s = p$ so that m uses the stronger p instead of its original parent $m.s$. In fact, the next definition and lemma show that one can perform more than one such subsumption operation in parallel, and the definition of subsumption can be slightly strengthened by considering the pivot variable. Indeed, Recycle-Units as listed in Algorithm 1 performs such substitutions with an arbitrary order.

Definition 2 (e-subsumption). *Let P be an e-resolution proof. We say that the node p e-subsumes the k nodes $m_1, \ldots, m_k$ with sides $s_1, \ldots, s_k$ (i.e., $s_i \in \{L, R\}$), if the following conditions hold for every i: The node m_i has two parents; m_i is not an ancestor of p; and*

$$p.C \leq \begin{cases} m_i.L.C \cup \{m_i.piv\} & \textit{if } s_i = L \\ m_i.R.C \cup \{\neg m_i.piv\} & \textit{if } s_i = R \,. \end{cases}$$

Lemma 2 (Parallel e-subsumption). *Let P be an e-resolution unsatisfiability proof, and let $p, m_1, \ldots, m_k$, $s_1, \ldots, s_k$ be as above. Then the proof P' obtained by setting $m_i.s_i = p$ for all i is an e-resolution unsatisfiability proof with $A(P') \subseteq A(P)$.*

Proof. Let P, P' be as above. We first argue that P' contains no cycles. Indeed, assume such a cycle exists. Obviously the cycle must contain some new edge. However, since all the new edges emanate from p, we can assume that the cycle contains exactly one new edge, say the edge from p to m_1. However, the rest of the cycle is a path from m_1 to p, which is a contradiction to the assumption that m_1 is not an ancestor of p.

Second, we argue that P' is an e-resolution unsatisfiability proof. The proof is by induction on the height of n with the induction claim: $n.C' \leq n.C$. Here we let $n.C, n.C'$ denote the consequent clauses at the node n when applying the e-resolution inference rule for the proof P, P' respectively. As the claim trivially holds for root nodes and for nodes with in-degree one, we can restrict our attention to some node n with two parents. By induction $n.L.C' \leq n.L.C \cup \{n.\text{piv}\}$ and $n.R.C' \leq n.R.C \cup \{\neg n.\text{piv}\}$.[4] Therefore, by (2) we conclude that $n.C' \leq n.C$ as claimed.

(Proof of Theorem 1)

[4] Note that the stronger claim $n.L.C' \leq n.L.C$ may not be correct since n.piv may have been added to $n.L.C$ if $n.L$ is one of the k nodes modified in the transition from P to P'.

Proof. The claim follows by applying Lemma 2 once for every iteration starting in line 2 of the RECYCLE-UNITS algorithm, and subsequently applying Lemma 1 once.

The only point that needs some extra elaboration is that the assumptions of Lemma 2 hold each time it is applied. Indeed, the algorithm RECYCLE-UNITS only considers the case of subsumption by a unit clause u. Furthermore, the algorithm checks for the condition $u = n.\text{piv}$ if $s_i = L$ and $u = \neg n.\text{piv}$ if $s_i = R$, which is strictly stronger than required by the Lemma.

5.3 The RECYCLE-PIVOTS Algorithm

This algorithm performs a DFS of the given e-resolution proof P starting at its sink P.sink. The main idea behind the algorithm is the following. Consider some internal node n with in-degree two. Then, by definition of an e-resolution proof, the clause $n.C$ will never contain the literal n.piv or $\neg n$.piv. This means, that as far as n is concerned, the clauses of all the ancestors of $n.L$ may contain n.piv, and all the ancestors of $n.R$ may contain $\neg n$.piv. Therefore, as far as n is concerned, any ancestor m of $n.L$ with $m.\text{piv} = n.\text{piv}$ can be simplified by setting $m.R = \text{nil}$. A similar statement holds if m is an ancestor of $n.R$..

To facilitate discussion, we introduce some notation. Given a proof P and a path $p = (n_1, n_2, \ldots, n_k)$ in the graph, where n_i is the parent of n_{i+1} on side $s_i \in \{L, R\}$, we define $\rho(n_i, n_{i+1})$ as n_{i+1}.piv if $s_i = R$ and as $\neg n_{i+1}$.piv if $s_i = L$. We define $\rho(p)$ as the union on i of $\rho(n_i, n_{i+1})$, and $\rho(n)$ as the intersection of $\rho(p)$ for all paths p from n to the sink. We use the notation $\neg\rho(n)$ for negation of the literal in $\rho(n)$, i.e. $\{\neg l : l \in \rho(n)\}$. Also, it can be easily verified that the following relations hold:

$$\rho(n.R) \subseteq \rho(n) \cup \{n.\text{piv}\}; \quad \rho(n.L) \subseteq \rho(n) \cup \{\neg n.\text{piv}\}. \tag{3}$$

The proof of Theorem 2 immediately follows from the following two lemmas:

Lemma 3. *The set RL on line 5 of algorithm* RECYCLE-PIVOTS *is always a subset of $\rho(n)$.*

Lemma 4. *Given an e-resolution proof P, let $P' =$* RECYCLE-PIVOTS(P)*. Then for any node n let C be the clause $n.C$ in P. Then the clause C' corresponding to $n.C$ in P' satisfies the relation $C' \setminus C \subseteq \neg\rho(n)$.*

(Proof of Lemma 3)

Proof. The proof is by induction on time. At time zero, $n = P$.sink and both RL and $\rho(n)$ are empty. So, assume that at some time $t > 0$ we are considering the node n. Because the algorithm is performing a DFS starting at P.sink, and because the value of RL is always determined before performing the recursive call, its value is completely determined by the nodes on the path p from n to the sink in the DFS tree. Furthermore, because RL is set to empty whenever a node has more than one parent, its value depends only on the longest sub-path

p' of p starting at n on nodes with at most one parent. Easy inspection shows that RL $= \rho(p')$. Since p' is contained in all paths from n to the sink, RL is a subset of $\rho(n)$ as claimed.

(Proof of Lemma 4)

Proof. For every node n, we let $n.C$ (resp. $n.C'$) denote the clause for n after applying RECONSTRUCT-PROOF on P (resp. P'). Then we prove the claim by induction on the height h of n, which is the length of the longest path from a root to n. The statement trivially holds for $h = 0$, since then $n.C = n.C'$. So, let n be some node of height $h > 0$. The proof proceeds by analyzing the possible actions of the algorithm on node n: i) lines 6-7, ii) line 9-10, and iii) lines 12-13.

The first case is proven using the definition of e-resolution (2), the induction hypothesis on $n.L$ and $n.R$, the relation (3), and the fact that neither n.piv nor $\neg n$.piv are in $\rho(n)$:

$$\begin{aligned} n.C' &= \text{e-resolution}_{n.\text{piv}}(n.L.C', n.R.C') \\ &\subseteq \text{e-resolution}_{n.\text{piv}}(n.L.C \cup \neg\rho(n.L), n.R.C \cup \neg\rho(n.R)) \\ &= \text{e-resolution}_{n.\text{piv}}(n.L.C, n.R.C) \cup \neg\rho(n) \\ &= n.C \cup \neg\rho(n). \end{aligned}$$

For the second case $n.\text{piv} \in \rho(n)$, we have:

$$n.C' = n.R.C' \subseteq n.R.C \cup \neg\rho(n.R) \subseteq n.C \cup \neg\rho(n).$$

The proof for the third case is analogous to the second. Therefore, $n.C' \subseteq n.C \cup R$ for all nodes n as claimed.

(Proof of Theorem 2)

Proof. The Theorem follows by applying Lemma 4 to the sink node of P. Since the sink R is the empty set and C is the empty clause, it follows that C' is the empty clause as well.

6 Experimental results and conclusions

We ran our linear reductions on the IBM benchmark suite, which comprises 63 different designs. On each design we ran bounded model checking with a bound k initially set to 10 and then incremented by 5, up to $k = 100$ or a bug was found. This gave us 630 unsatisfiable instances. From those we chose only the instances that take 10 seconds or more for RUN-TILL-FIX.[5] This left us with 67 proofs. We set the timeout for each run to be 1800 seconds. We used a 64-bit machine with 8 GB memory, and 2x2.4Ghz Opteron dual core.

[5] This creates a bias against run-till-fix, but recall that we are not competing against run-till-fix – we only check whether our methods can be helpful when run-till-fix fails with a short time-out.

		Leaves				Nodes			
Reduction	Time	Before	After	Per sec	Ratio	Before	After	per sec	Ratio
RUN-TILL-FIX	8095	1002924	533941	57.9	0.53	11830898	17677419	-722.2	1.49
units	1002.5	1002924	997674	5.2	0.99	11830898	11513195	316.9	0.97
pivots	32.5	1002924	953585	1518.6	0.95	11830898	10464394	42059.2	0.88
units + pivots	1235.8	1002924	949279	43.4	0.95	11830898	10247401	1281.3	0.87

Fig. 6. Reduction in proof leaves and nodes. Run time is cumulative for 63 unsatisfiable runs. The 'Per sec' columns indicate the number of removed leaves / nodes per second. The 'Ratio' columns indicate the ratio between the number of leafs (or nodes) before and after the reduction.

The results appear in Fig. 6. What can be concluded from them is that the linear reductions we proposed are on one hand fast (especially RECYCLE-PIVOTS), but their effectiveness in reducing leaves is small: only $\approx 5\%$ of the leaves are removed, comparing to 47% with RUN-TILL-FIX. When it comes to the size of the proof itself, it turns out that RUN-TILL-FIX *increases* the number of proof nodes substantially (by 49%) whereas our linear reductions decrease their size by 13%. The size of the proof can be relevant when computing interpolants, and indeed as future work we intend to check its effectiveness on IBM's interpolation-based model-checker.

To conclude, we showed two techniques for fast preprocessing (perhaps it should be called postprocessing) of resolution proofs. They cannot replace RUN-TILL-FIX if the main goal is the reduction in core regardless of the time it takes, but they can complement it or even replace it in a realm of short time outs. As indicated above, it is more valuable in scenarios in which decreasing the proof size rather than its core is what matters.

References

1. N. Amla and K. McMillan. Automatic abstraction without counterexamples. In H. Garavel and J. Hatcliff, editors, *TACAS'03*, volume 2619 of *Lect. Notes in Comp. Sci.*, 2003.
2. N. Amla and K. L. McMillan. A hybrid of counterexample-based and proof-based abstraction. In *Formal Methods in Computer-Aided Design, 5th International Confrence, FMCAD 2004*, pages 260–274, 2004.
3. R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. A. Brady. Deciding bit-vector arithmetic with abstraction. In O. Grumberg and M. Huth, editors, *13th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, pages 358–372, 2007.
4. A. Cimatti, A. Griggio, and R. Sebastiani. A simple and flexible way of computing small unsatisfiable cores in sat modulo theories. In *SAT*, pages 334–339, 2007.
5. N. Dershowitz, Z. Hanna, and A. Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *SAT*, volume 4121 of *Lecture Notes in Computer Science*, pages 36–41. Springer, 2006.

6. R. Gershman, M. Koifman, and O. Strichman. Deriving small unsatisfiable cores with dominators. In *Proc. 18th Intl. Conference on Computer Aided Verification (CAV'06)*, number 4144 in Lect. Notes in Comp. Sci., pages 109–122, 2006.
7. R. Gershman and O. Strichman. Haifasat: A new robust SAT solver. In Y. W. Shmuel Ur, Eyal Bin, editor, *First International Haifa Verification Conference*, volume 3875 of *Lect. Notes in Comp. Sci.*, pages 76 – 89. Springer-Verlag, 2005.
8. É. Grégoire, B. Mazure, and C. Piette. Local-search extraction of muses. *Constraints*, 12(3):325–344, 2007.
9. O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation-widening for multi-process systems. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages*, pages 122–131. ACM Press, 2005.
10. J. Huang. Mup: A minimal unsatisfiability prover. In *Proc. of the 10th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 432–437, 2005.
11. D. Kroening, J. Ouaknine, S. Seshia, and O. Strichman. Abstraction-based satisfiability solving of Presburger arithmetic. In R. Alur and D. Peled, editors, *Proc. 16th Intl. Conference on Computer Aided Verification (CAV'04)*, number 3114 in LNCS, pages 308–320, Boston, MA, July 2004. Springer-Verlag.
12. T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.
13. I. Lynce and J. Marques-Silva. On computing minimum unsatisfiable cores. In *Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, pages 305–310, 2004.
14. K. McMillan. Interpolation and sat-based model checking. In J. Warren A. Hunt and F. Somenzi, editors, *cav03*, Lect. Notes in Comp. Sci., Jul 2003.
15. M. N. Mneimneh, I. Lynce, Z. S. Andraus, J. P. M. Silva, and K. A. Sakallah. A branch-and-bound algorithm for extracting smallest minimal unsatisfiable formulas. In *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 467–474. Springer, 2005.
16. Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. Amuse: a minimally-unsatisfiable subformula extractor. In *DAC '04*, pages 518–523, 2004.
17. C. H. Papadimitriou and D. Wolfe. The complexity of facets resolved. *J. Comput. Syst. Sci.*, 37(1):2–13, 1988.
18. L. Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004.
19. O. Shtrichman. Prunning techniques for the SAT-based bounded model checking problem. In *proc. of the 11th Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, Edinburgh, Sept. 2001.
20. G. Tseitin. On the complexity of proofs in poropositional logics. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning: Classical Papers in Computational Logic 1967-1970*, volume 2. Springer-Verlag, 1983. Originally published 1970.
21. J. Whittemore, J. Kim, , and K. Sakallah. Satire: a new incremental satisfiability engine. In *In IEEE/ACM Design Automation Conference (DAC)*, 2001.
22. L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, 2001.
23. L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas, 2003.

A SAT solvers cannot produce all resolution proofs

Modern DPLL solvers that restrict learning to asserting clauses (virtually all competitive solvers that we are aware of) are restricted in the resolution proofs that they can generate. In asserting clauses there is exactly one literal from the decision level in which the conflict occurred. Recall that learned clauses are the internal nodes in the resolution.

On the other hand, some pairs of literals can be forced by the constraints to be assigned in the same level. Consider, for example, the clause set

$$(-1\ 2)\ (1\ \ -2)\ (-2\ 3)\ (2\ \ -3)$$

forcing $1, 2$ and 3 to be equal. Resolving the first and third clauses yields $C = (-1\ 3)$. But since these variables imply each other, they must have the same decision level, which means that a clause such as C above cannot be an asserting clause.

The proof reconstruction can result in a proof of a clause such as C, however. Assume we also have the clauses $(-1\ 4), (-4\ 6), (4\ 5)$ and $(3\ -4\ -6)$, and that the proof graph includes the subgraph depicted in Fig. 7a. The other two graphs show the process of applying RecyclePivots, ending with a proof of C. RecyclePivots detects that the pivot variable 4 is used in both C7 and C4, removes the edge $(C1, C4)$, copies C2 to C4, and continues from there as shown in drawings b and c in the same figure.

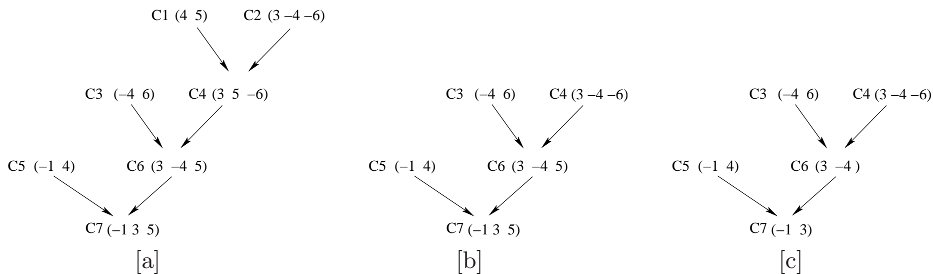

Fig. 7. Recycling pivots can generate proofs that cannot be generated by a SAT solver.

The restriction to asserting clauses does not restrict the power of SAT solvers, at least not according to the example above. The reason is that for each clause that cannot be asserting according to the pattern above, there is an asserting clause that subsumes it that is still implied by the formula. In the example above rather than learning C, an asserting clause can be a unit (-1) or (3). If C is implied, then so is each of these two clauses.