An Efficient and Flexible Approach to Resolution Proof Reduction

Simone Fulvio Rollini, Roberto Bruttomesso, and Natasha Sharygina

University of Lugano, Formal Verification Group, Lugano, Switzerland
{simone.fulvio.rollini,roberto.bruttomesso,natasha.sharygina}@usi.ch

Abstract. A resoution proof is a certificate of the unsatisfiability of a Boolean formula. Resolution proofs, as generated by modern SAT solvers, find application in many verification techniques. For efficiency smaller proofs are preferable over larger ones. This paper presents a new approach to proof reduction, situated among the purely post-processing methods. The main idea is to reduce the proof size by eliminating *redundancies* of occurrences of pivots along the proof paths. This is achieved by matching and rewriting *local contexts* into simpler ones. In our approach, rewriting can be easily customized in the way local contexts are matched, in the amount of transformations to be performed, or in the different application of the rewriting rules. We provide an extensive experimental evaluation of our technique on a set of benchmarks, which shows considerable reduction in the proofs size.

1 Introduction

A propositional proof of unsatisfiability is a certificate of the unsatisfiability of a Boolean formula. It is straightforward to instruct a state-of-the-art solver based on DPLL to return proofs: a resolution proof, in particular, can be derived by logging the resolution steps during conflict analysis [17].

Resolution proofs, as generated by modern SAT solvers, find application in many verification techniques. For instance, Amla and McMillan's [3] method for automatic abstraction uses proofs of unsatisfiability derived from SAT-based bounded model checking as a guide for choosing an abstraction for unbounded model checking. Proofs can be used as justifications of specification of inconsistency in various industrial applications (e.g. automotive industry [15]). Another noteworthy application of proofs is in the context of interpolation-based model checking [9, 11, 12]. Moreover, a proof can be used to extract an unsatisfiable core, an inconsistent subset of clauses.

For most applications, large and redundant proofs negatively affects efficiency. Unfortunately the size of a propositional resolution proof may grow exponentially w.r.t. the input. Even when the proofs have a manageable size, further speed-ups may be achieved with proof-compression techniques. Several reduction techniques are known in literature. We briefly recall them as follows. Amjad [1] suggests a heuristic reordering of the resolution steps, based on literals linking and subsumption checking. Sinz [14] explicitly assumes a DPLL context

and focuses on identifying and merging shared substructures in the derivation of learned clauses obtained from conflict graphs. Amjad [2] develops this approach by introducing memoization of common subproofs, in a string data compression perspective; Cotton [6] also adopts a memoization technique, as well as a rewriting procedure based on variable valuation. Bar-Ilan et al. [4] present a post-processing reduction technique based on discovering resolutions on the same pivot along paths in the proof and on maximizing the reuse of derived unit clauses in the proof. In the context of proofs and interpolants, D'Silva et al. [7] introduce a transformation framework to reorder a proof w.r.t. a partial order among pivots, with reduction being a side effect for some benchmarks.

This paper presents another approach for proof reduction. It is situated among the purely proof post-processing techniques, and it is independent from the way the refutation is produced: the algorithm we propose can be applied to an arbitrary propositional resolution proof of unsatisfiability.

Resolution proofs can be thought of as trees, where the leaves are the original clauses in the problem and the root is the empty clause. The main idea behind our approach is to reduce the proof size by eliminating *redundancies* in the proof. This idea has been already exploited by other authors, for instance in [4], where the proof tree is analyzed to detect multiple resolution steps on the same variable along a path from the root to the leaves. (more details are given in §2).

In particular our approach aims at detecting redundancies by matching and rewriting *local contexts* in the proof into simpler ones. The rewriting process can be easily customized in the way local contexts are matched, in the amount of transformations to be performed, or in the different application of the rewriting rules. It results in a considerable reduction of the proof size as confirmed by our experimental evaluation of the new technique.

Our method inherits the idea of matching contexts in a proof from [5], where the application of the rules is aimed at reordering pivots for deriving interpolants. We observed that a side-effect of some pivot reordering steps may result in a reduction of the proof tree. This paper generalizes this effect to achieve systematic proof compression.

We compare our approach with that of [4], by pointing out similarities and discrepancies, strengths and weaknesses of both. We illustrate a simple way of combining them in a unified and effective strategy. We provide an extensive experimental evaluation on a set of SMT benchmarks, for which the presented techniques are very effective.

The paper is organized as follows. §2 recalls some notions about resolution and introduces the notation; it then goes over the set of transformation rules presented in [5] and the RecyclePivots algorithm of [4]. §3 describes the new proof transformation algorithm. §4 proposes a combined strategy of our approach with that of [4]. §5 illustrates the results of a number of experiments on a set of SMT benchmarks, showing the individual performances of the two algorithms and the benefits that can be gained by merging them into a combined procedure. §6 draws the conclusions.

2 Preliminaries and Previous Work

In the following we shall use o, p, q, r (possibly with subscripts) to denote Boolean variables, s, t to denote literals, $\alpha, \beta, \ldots$ to denote clauses, and $C, D, \ldots$ to denote sub-clauses. The empty clause is denoted by $\bot$. We will write clauses as lists of literals and sub-clauses, omitting the "$\vee$" symbol, as for instance $p\overline{q}C$. We will use the form $\alpha \subseteq \beta$ to indicate that α *subsumes* β, i.e., that the set of literals α is a subset (not necessarily proper) of the set of literals β. Also we will assume that clauses do not contain duplicated literals or both the occurrences of a literal and its negation. If s is a literal we use $var(s)$ to denote the variable associated with it.

Resolution is the following proof rule:

$$\frac{p\,C \qquad \overline{p}D}{CD}\,p$$

Clauses pC and $\overline{p}D$ are called *antecedents*, CD is the *resolvent*, and p is the *pivot* variable. Throughout the paper we shall use the notion of *resolution proof*.

Definition 1 (Resolution Proof). *A* resolution proof *of a clause λ from a set of clauses $\mathbb{S}$ is a tree such that (i) its leaves are clauses in $\mathbb{S}$, (ii) the root is λ, (iii) intermediate clauses are derived by means of an application of the resolution rule.*

From now on we shall focus on the notion of propositional resolution proof, and we will just use the term "proof" for brevity. We say that a proof is a *refutation* or a *proof of unsatisfiability* if $\lambda \equiv \bot$. In real-world applications proofs are rarely stored as trees. For instance proofs generated by DPLL SAT-Solvers are normally stored as DAGs (Directed Acyclic Graph), in order to minimize the memory consumption. We therefore introduce the following alternative notion of resolution proof, which is equivalent to the previous one but more suitable for describing the graph-based transformation algorithms presented in this paper.

Definition 2 (Graph-based Resolution Proof). *A* resolution proof *of a clause λ from a set of clauses $\mathbb{S}$ is a Directed Acyclic Graph $G(V, E)$ such that:*

(i) Each node $n \in V$ is associated with a clause n_{cl}.
*(ii) Each node has either no antecedents (*leaf *node) or it has exactly two antecedents from which it is derived by means of an application of the resolution rule (*derived *node).*
(iii) For each pair of nodes m and n there is a directed edge $m \rightarrow n \in E$ if and only if m_{cl} is an antecedent of n_{cl} (by extension, we will call m an antecedent of n and n a resolvent of m).
(iv) For each derived node n, we denote with n_{piv} the pivot of the resolution step of which n_{cl} is the resolvent, while n^l and n^r are the left and right antecedents (we assume they respectively contain the positive and negative occurrence of the pivot).

Let n be a node in a proof P. The portion of P that is backward-reachable from n is clearly a proof of n_{cl}. A proof is said to be *regular* [16] if each variable is used as a pivot at most once along each path. A proof is *tree-like* if in the corresponding DAG every node (apart from root and leaves) has exactly one resolvent.

Similarly to [4], we will allow a little flexibility in the notion of proof and distinguish between a *legal* and an *illegal* proof. A legal proof is simply a DAG as in Definition 2; an illegal proof is a graph which has undergone transformations in such a way that some nodes (clauses) might not be the resolvents of their antecedents clauses anymore. In this paper however any illegal proof represents an intermediate transformation step in the algorithm, and the proof can always be *reconstructed* into a legal one, as better explained in the next sections.

2.1 The RecyclePivots Approach

The RecyclePivots algorithm of [4] is based on the observation that, along each path from a leaf to the root, it is unnecessary to resolve upon a certain pivot more than once. Such redundancies can be removed, for example by keeping (for a given variable and a path) only the resolution step closest to the root, while cutting the others away.

Example 1. Consider the leftmost path of the proof in Figure 1a. Variable p is used twice as pivot. The topmost resolution step is redundant as it eliminates a variable (p) which is reintroduced in a subsequent step. A better proof can be achieved by eliminating the topmost resolution step, and by adjusting the proof accordingly. The resulting final proof is shown in Figure 1b.

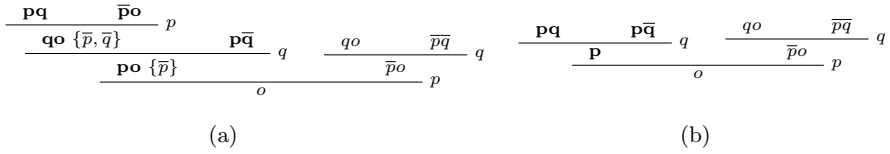


Fig. 1. RecyclePivots execution. Bold-faced font highlights the differences in the proofs. Curly brackets contain the set of removable literals.

As suggested by Example 1, redundancies are caused by the re-introduction of a previously eliminated variable in a path from a leaf to the root.

Algorithm 1 shows the recursive version of RecyclePivots. It works with a depth-first visit of the proof graph, from the root to the leaves. It receives as input a node of the proof (initially $\bot$) and a set of removable literals RL (initially empty). The removable literals are essentially the (partial) collection of pivot literals encountered during the exploration of a path, from the root to the leaves. If the pivot variable of the resolution step under consideration is in RL (lines 14

and 17), then the resolution step is redundant and one of the antecedents may be removed from the proof (lines 15 and 18)[1].

Notice that in the case of tree-like proofs the outcome of the process is a regular proof. For generic DAG-like proofs the algorithm is executed in a limited form (when multiple resolvents are detected) precisely by resetting RL (line 10); therefore the result is not necessarily a regular proof.

Algorithm 1: RecyclePivots(n,RL)

```
   Input: A node n, a set of removable literals RL
 1 begin
 2     if n is visited then
 3         return;
 4     else
 5         Mark n as visited;
 6         if n is a leaf then
 7             return;
 8         else
 9             if n has more than one resolvent then
10                 RL ← ∅;
11             if n_piv ∉ RL and ¬n_piv ∉ RL then
12                 RecyclePivots(n^l, RL ∪ {¬n_piv});
13                 RecyclePivots(n^r, RL ∪ {n_piv});
14             else if n_piv ∈ RL then
15                 n^l ← null;
16                 RecyclePivots(n^r, RL);
17             else if ¬n_piv ∈ RL then
18                 n^r ← null;
19                 RecyclePivots(n^l, R);
20 end
```

2.2 Local Proof Transformation Rules

The reduction approach proposed in this paper is built upon a generic proof transformation framework presented in [5], that we recall as follows. The framework is based on a set of rewriting rules that transform a subproof into an equivalent or stronger one. Each rewriting rule is defined to match a particular *context*, identified by two consecutive resolution steps (see Figure 2).

A context involves two pivots p and q and five clauses $\alpha, \beta, \gamma, \delta, \eta$. Clearly p is contained in β and γ (with opposite polarity), and q is contained in δ and α (with opposite polarity).

[1]The resulting illegal proof has to be reconstructed to a legal one. It can be done in linear time [4].

$$\dfrac{\dfrac{\beta \qquad \gamma}{\delta}\, p \qquad \alpha}{\eta}\, q$$

Fig. 2. A rule context.

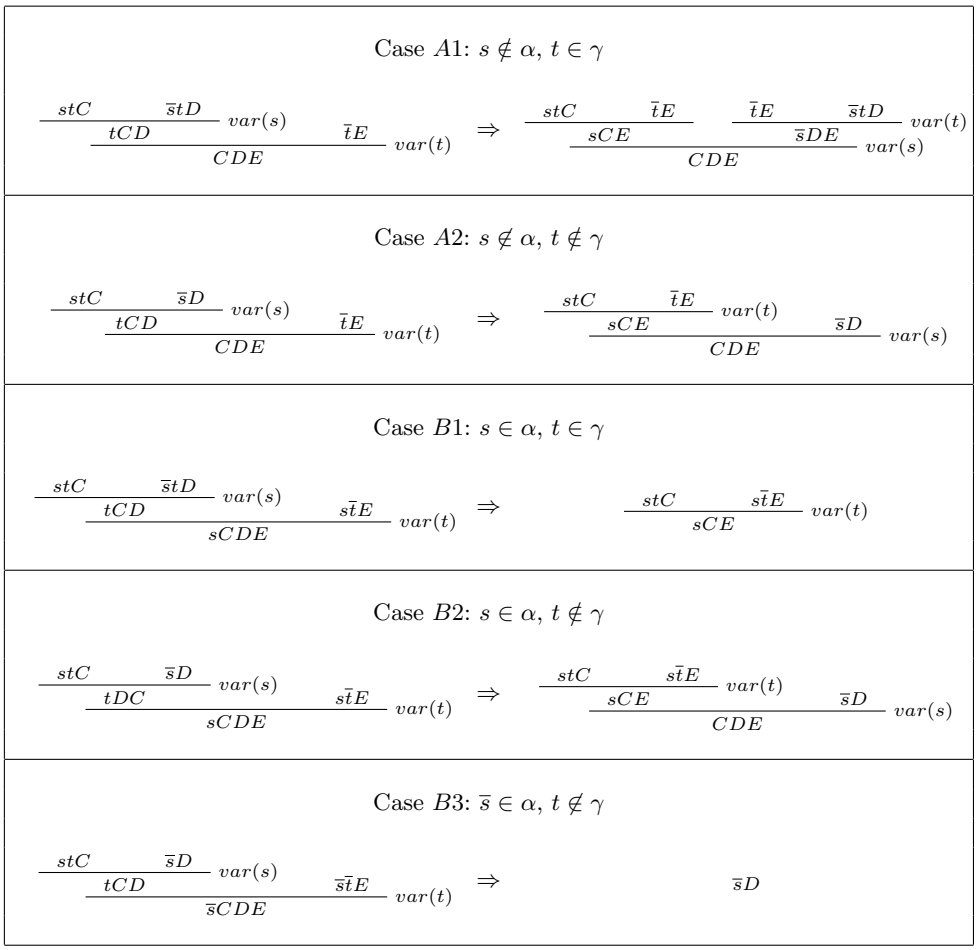


Fig. 3. Local transformation rules.

Figure 3 shows the set of five proof transformation rules introduced in [5][2]. The set is exhaustive w.r.t. all the possible local contexts in a proof, modulo symmetry and the sign of s and t (notice that they are both literals). In [5] we used the set of rules to compute interpolants in the context of SMT. In short, in

[2]The proof transformations associated with the rules $A1$ and $A2$ were discussed first in [10] and later in [8].

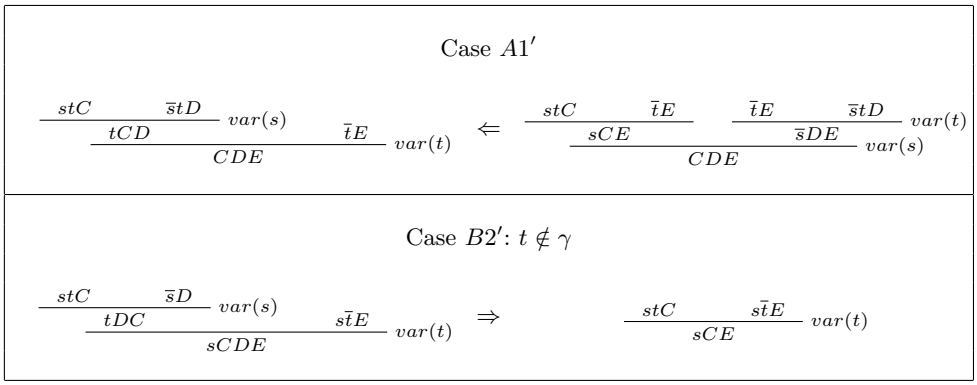


Fig. 4. Two new reduction rules.

order to apply state-of-the-art solving methods for interpolation, it is necessary to reorder the pivots in the proof according to some partial order. This reordering can be achieved with an exhaustive applications of the rules in Figure 3: each rule either lifts t over s, or it eliminates their alternation, until the proof respects the given partial order.

The reader may notice that some rules, those with prefix B, have the side-effect of reducing the complexity of the proof: in all B cases, in fact, the root of the right-hand side of the rule is *stronger* than the root of the left-hand side, even though it is derived from the same set of premises. In this paper we systematically exploit this reduction effect to the aim of reducing the proof as much as possible.

Example 2. Recall Figure 1a from Example 1. The context highlighted by the bold-faced font corresponds to the premise of our rule $B2$. Proof is again rewritten as in Figure 1b.

3 Reduction by Proof Manipulation

This section presents a new algorithm for proof reduction, which is based on the rules recalled in §2.2 and on a couple of new reduction rules. We start by introducing the new rules and then we present a deterministic application strategy aimed at proof reduction. Then we compare with the Recycle_Pivots approach of [4] and finally we draw a simple and effective way to combine the two approaches.

3.1 Two New Reduction Rules

Although the set of contexts for rules in Figure 3 is exhaustive, we noticed that in the case of $B2$ another transformation can be performed; a new reduction rule $B2'$, associated with the same context of $B2$, has thus been introduced here

for the sake of completeness. Another reduction rule $A1'$ is conceived as the "inverse" of $A1$ (notice the direction of the arrow).

3.2 The Transformation Algorithm

Recall that in [5] the rewriting rules were employed to perform a local reordering of the pivots. In particular $A1$, $A2$, and $B2$ were used to swap the position of two subsequent pivots, while $B1$ and $B3$ were employed to eliminate a re-introduction of a pivot variable.

Here the focus is on reduction: rules $B1$, $B2$, $B2'$ and $B3$, when applied to a context, are directly responsible for the reduction of the proof, as their root η' is stronger than that of the original context η, i.e. $\eta' \subset \eta$. Notice that after the application of a B rule the proof might become *illegal*, as some literals in $\eta \setminus \eta'$ might be involved in another resolution step along the path to the root. We shall explain this situation by means of an example. Consider the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathbf{pq} \quad \mathbf{\overline{p}o}}{\mathbf{qo}}\,p \quad \mathbf{p\overline{q}}}{\mathbf{po}}\,q \quad \dfrac{qr \quad \overline{pq}}{\overline{p}r}\,q}{or}\,p \quad \overline{o}s}{rs}\,o \tag{1}$$

The highlighted context can be reduced via an application of $B2$ as follows:

$$\dfrac{\dfrac{\dfrac{\mathbf{pq} \quad \mathbf{p\overline{q}}}{\mathbf{p}}\,q \quad \dfrac{qr \quad \overline{pq}}{\overline{p}r}\,q}{or}\,p \quad \overline{o}s}{rs}\,o \tag{2}$$

The proof has become illegal as the literal o is now not introduced by any clause ($o \in \{p, o\} \setminus \{p\}$). Since we have derived a stronger conclusion ($\{p\} \subset \{p, o\}$) o is now redundant and it can be eliminated all the way down to the root or up to the point it is reintroduced by some other resolution step. In this example we can safely remove o together with the last resolution step which also becomes redundant. The resulting legal (and stronger) proof becomes:

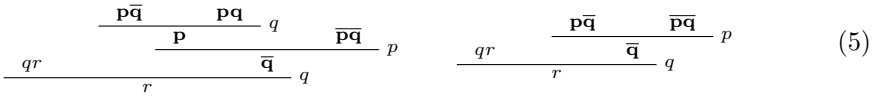
$$\dfrac{\dfrac{pq \quad p\overline{q}}{p}\,q \quad \dfrac{qr \quad \overline{pq}}{\overline{p}r}\,q}{r}\,p \tag{3}$$

At this stage no other B rule could be directly applied to the proof.

Rule $A2$ does not perform any reduction on its own. However it is still used in our framework. Its contribution is to produce a "shuffling" effect in the proof, in order to create more chances for the B rules to be applied. Rule $A1$ instead is never used (it may increase the size of the proof). Consider again our running example. An application of $A2$ can be executed as follows:

$$\dfrac{\dfrac{pq \quad p\overline{q}}{\mathbf{p}}\,q \quad \dfrac{\mathbf{qr} \quad \mathbf{\overline{pq}}}{\mathbf{\overline{p}r}}\,q}{\mathbf{r}}\,p \qquad \dfrac{\dfrac{p\overline{q} \quad pq}{\mathbf{p}}\,q \quad \mathbf{\overline{pq}}}{\mathbf{\overline{q}}}\,p \quad \mathbf{qr}}{\mathbf{r}}\,q \tag{4}$$

The application of $A2$ has now exposed a new redundancy involving the variable q. The proof can be readily simplified by means of the application of $B2'$ as follows:



(5)

The rewriting framework defined by our rules leaves to the user the flexibility of choosing a particular *strategy* and a *termination criterion* for their application. A naive strategy is to eagerly try the application of the B rules until possible, and then try to shuffle the proof by means of $A2$, in the hope of disclosing other redundancies, and apply B rules again. However there is usually a huge number of contexts where $A2$ could be applied, and it is computationally expensive to predict whether one or a chain of applications will eventually lead to the creation of contexts for a B rule (more details on the heuristics used are given in the next section).

Therefore there are two obvious termination criteria that could be used: to stop when a *timeout* is reached or after a certain number of contexts has been explored. In our approach we set both limits, and stop whenever the first has been reached. Our reduction procedure is ReduceAndReconstruct, listed as Algorithm 2.

Algorithm 2: ReduceAndReconstruct

Input: A proof; *timelimit*: a timeout; *numloops*: the number of transformation loops to perform
Output: A reduced (legal) proof

```
1 begin
2     for i=1 to numloops do
3         ReduceAndReconstructLoop( );
4         if timeout then
5             break;
6     end
7 end
```

The procedure is based on the routine ReduceAndReconstructLoop( ), listed as Algorithm 3.

The algorithm combines proof reduction and reconstruction (due to removal of literals as explained above). It works as follows. At first it performs a topological ordering of the graph (line 2), in order to ensure that each node is visited after its antecedents. Then it analyzes one node at a time, checking if the corresponding resolution step is still valid (line 5). If the resolution step is valid, it updates the resolvent clause, determining the node contexts (if any) and the associated rules. At most one rule is applied, and the decision is based on local heuristic considerations (see next section). If the resolution step is not valid

Algorithm 3: ReduceAndReconstructLoop( )

```
   Input: A proof
   Output: A legal proof
   Data: TS: nodes topological sorting vector
 1 begin
 2     TS ← topological_sorting(proof);
 3     foreach n ∈ TS do
 4         if n is not a leaf then
 5             if n_piv ∈ n^l_cl and ¬n_piv ∈ n^r_cl then
 6                 n_cl ← Res(n^l_cl, n^r_cl);
 7                 Determine left context of n, if any;
 8                 Determine right context of n, if any;
 9                 Heuristically choose one context (if any) and apply the
                   corresponding rule;
10             else if n_piv ∉ n^l_cl and ¬n_piv ∈ n^r_cl then
11                 Substitute n with n^l;
12             else if n_piv ∈ n^l_cl and ¬n_piv ∉ n^r_cl then
13                 Substitute n with n^r;
14             else if n_piv ∉ n^l_cl and ¬n_piv ∉ n^r_cl then
15                 Heuristically choose an antecedent n^l or n^r;
16                 Substitute n with n^l or n^r;
17     end
18 end
```

and either antecedent does not contain the pivot (lines 10, 12, 14), the step is removed, by replacing the resolvent with that antecedent (which, missing the pivot, subsumes the resolvent itself); at graph level, n is substituted with n^l or n^r. Notice that the antecedent not responsible for the substitution might have lost all its resolvents and thus it does not contribute to the proof anymore; in that case it is pruned away, together with the portion of the subgraph rooted in it which has become unreachable.

We have the following result.

Theorem 1 *ReduceAndReconstructLoop( ) outputs a legal proof.*

3.3 Duplications and Heuristics

If the input proof is not tree-like, then the clause δ of a context may participate in more than one resolution step: in this case the modifications to the proof graph cannot be executed *in place*. In fact, if δ, associated with a node n_δ, is part of a context to be transformed, it is necessary to create a copy n'_δ of n_δ, in order to preserve the correctness of the other resolution steps (this problem does not affect clauses n_β, n_γ, n_α of a context). Notice that a duplication increases the size of the proof. Therefore rewriting steps that generate duplications are carried out under some restrictions. In particular we allow duplications only in the case of rules $B1$, $B2$, $B2'$, $B3$.

As far as the heuristics for choosing the application of a rule are concerned, we respect the following precedence order ($X > Y$ means: the application of X is preferred over that of Y):

$$B2 > B3 > \{B2', B1\} > A1' > A2$$

For a more thorough discussion of heuristics in combination with duplications we refer the reader to the appendix.

4 Comparison with RecyclePivots and Combination

As we have seen both RecyclePivots and ReduceAndReconstruct aim at reducing the proof by exploiting redundancies in the pivots along a path from the root to the leaves. The main difference between the approaches is that RecyclePivots operates on a *global context* without changing the topology of the proof, while ReduceAndReconstruct operates on *local contexts* and it allows the topology to change. Both approaches have positive and negative aspects.

Operating on a global context without changing the topology allows a one-pass visit and reduction of the proof. Maintaining a fixed topology of the proof however may prevent the disclosure of hidden redundancies. For instance the application of RecyclePivots to our previous running example would have stopped to step (3), since no more redundant pivots can be found along a path (the proof is regular). Our local contexts instead have to be gathered and considered multiple times. On the other hand, the ability of ReduceAndReconstruct to change the topology may allow more redundancies to be exposed.

Another advantage of RecyclePivots is that it can reduce redundancies that are separated by many resolution steps. Our B rewriting rules instead are applicable only when there is a re-introduction of a certain variable immediately after a resolution upon it (s in Figure 3-4). Such configurations, when not present in the proof, can be produced by means of the application of the $A2$ rule. It is not clear whether RecyclePivots could be simulated with a particular application strategy of our rules. We leave this investigation as future work.

In an attempt of exploiting the advantages of both approaches, we devised a naive hybrid approach of both reduction strategies, listed as Algorithm 4.

5 Experiments

We carried out an evaluation of the three algorithms RecyclePivots (RP), ReduceAndReconstruct (RR), and CombinedReduction (CR). The algorithms have been implemented inside the tool OpenSMT [13], with proof-logging capabilities enabled. Unfortunately we could not test our algorithms on the same instances of [4], which are covered by IP rights. However on purely (unsatisfiable) SAT instances taken from the SAT competition website[3], we observed a similar behavior for the three techniques (for lack of space we do not report on this data here).

[3] `http://www.satcompetition.org`

Algorithm 4: CombinedReduction

Input: A legal proof; *numextloops*: number of global loops; *numintloops*: number of transformation loops for each global loop; *timelimit*: a timeout

Output: A reduced legal proof

```
1 begin
2     timeslot = timelimit/numextloops;
3     for i=1 to numextloops do
4         RecyclePivots(n⊥,∅);
5         // RPtime is the time taken by RecyclePivots in the last call;
6         ReconstructAndTransform(timeslot − RPtime,numintloops);
7     end
8 end
```

We also experimented on the set of unsatisfiable benchmarks taken from the SMT-LIB[4], from the categories QF_UF, QF_IDL, QF_LRA, QF_RDL. For these sets of benchmarks we have noticed that the aforementioned reduction techniques are very effective. We believe that the reason is connected with the fact that the introduction of theory-lemmata in SMT is performed lazily: the delayed introduction of clauses involved in the final proof may negatively impact the online proof construction in the SAT-Solver.

All the experiments were carried out on an Ubuntu server featuring a Dual-Core 2GHz Opteron CPU and 4GB of memory; a timeout of 10 minutes and a memory threshold of 2GB (whatever is reached first) were put as limit to the executions.

The executions of RR and CR are parameterized with a time threshold, as the algorithms require. We have chosen to set the threshold as a fraction of the time taken by the solver to solve the benchmarks: more difficult instances are likely to produce larger proofs, and therefore more time is necessary to achieve reduction. Notice that, regardless of the ratio, RR and CR both perform at least one complete transformation loop, which could result in an execution time slightly higher than expected for low ratios and small proofs.

Table 1 shows the average reduction of the proofs after the application of the algorithms[5]. Table 1a shows the reduction obtained after the execution of RecyclePivots. Table 1b instead shows the reduction obtained with Reduce-AndReconstruct and CombinedReduction parameterized with timeout (ratio · solving-time). In particular we report the reduction in the number of nodes and edges, the reduction of the unsatisfiable core, and the actual transformation time. Table 2 is organized as Table 1 except that it reports the best reduction values obtained over all the benchmarks suites.

[4] `http://www.smt-lib.org`

[5] Full experimental data, as well as executables used in tests are available at `http://www.inf.usi.ch/phd/rollini/hvc.html`.

	NumBench	AvgRedNodes	AvgRedEdges	AvgRedCore	AvgTranTime (s)
RP	1370	6.7%	7.5%	1.3%	1.7

(a)

	NumBench		AvgRedNodes		AvgRedEdges		AvgRedCore		AvgTranTime (s)	
Ratio	RR	CR	RR	CR	RR	CR	RR	CR	RR	CR
0.01	1364	1366	2.7%	8.9%	3.8%	10.7%	0.2%	1.4%	3.5	3.4
0.025	1363	1366	3.8%	9.8%	5.1%	11.9%	0.3%	1.5%	3.6	3.6
0.05	1364	1366	4.9%	10.7%	6.5%	13.0%	0.4%	1.6%	4.3	4.1
0.075	1363	1366	5.7%	11.4%	7.6%	13.8%	0.5%	1.7%	4.8	4.5
0.1	1361	1364	6.2%	11.8%	8.3%	14.4%	0.6%	1.7%	5.3	5.0
0.25	1357	1359	8.4%	13.6%	11.0%	16.6%	0.9%	1.9%	8.2	7.6
0.5	1346	1348	10.4%	15.0%	13.3%	18.4%	1.1%	2.0%	12.1	11.5
0.75	1339	1341	11.5%	16.0%	14.7%	19.5%	1.2%	2.1%	15.8	15.1
1	1335	1337	12.4%	16.7%	15.7%	20.4%	1.3%	2.2%	19.4	18.8

(b)

Table 1. Results for SMT benchmarks. NumBench reports the number of benchmarks solved and processed within the time/memory constraints, AvgRedNodes and AvgRedEdges report the reduction in the number of nodes and edges of the proof graphs, and AvgRedCore reports the average reduction in the unsatisfiable core size. AvgTranTime is the average transformation time in seconds.

On a single run RP clearly achieves the best results for reduction with respect to transformation time. To get the same effect on average on nodes and edges, for example, RR needs about 5 seconds and a ratio transformation time/solving time equal to 0.1, while RP needs less than 2 seconds. As for core compression, the ratio must grow up to 1. On the other hand, as already remarked, RP cannot be run more then once.

The combined approach CR shows a performance which is indeed better than the other two algorithms taken individually. it is interesting to see that the global perspective adopted by RP gives an initial substantial advantage, which is slowly but constantly reduced as more and more time is dedicated to local transformations and cuts.

Table 2b displays some remarkable peaks of reduction obtained with RR and CR approaches on the best individual instance. Interestingly we have noticed that in some benchmarks, like 24.800.graph.smt of QF_IDL suite, RecyclePivots does not achieve any reduction, due to the high amount of nodes with multiple resolvents present in its proof that forces RecyclePivots to reset the relevant literal set RL all the time. CombinedReduction instead, even for a very small ratio (0.01), performs very well (47.6% reduction for nodes, 49.7% reduction for edges and 45.7% for core).

	MaxRedNodes	MaxRedEdges	MaxRedCore
RP	65.1%	68.9%	39.1%

(a)

	MaxRedNodes		MaxRedEdges		MaxRedCore	
Ratio	RR	CR	RR	CR	RR	CR
0.01	54.4%	66.3%	67.7%	70.2%	45.7%	45.7%
0.025	56.0%	77.2%	69.5%	79.9%	45.7%	45.7%
0.05	76.2%	78.5%	78.9%	81.2%	45.7%	45.7%
0.075	76.2%	78.5%	79.7%	81.2%	45.7%	45.7%
0.1	78.2%	78.8%	82.9%	83.6%	45.7%	45.7%
0.25	79.3%	79.6%	84.1%	84.4%	45.7%	45.7%
0.5	76.2%	79.1%	83.3%	85.2%	45.7%	45.7%
0.75	78.2%	79.9%	84.4%	86.1%	45.7%	45.7%
1	78.3%	79.9%	84.6%	86.1%	45.7%	45.7%

(b)

Table 2. Results for SMT benchmarks. MaxRedNodes and MaxRedEdges are the maximum reduction of nodes and edges achieved by the algorithms in the suite on a single benchmark. The benchmark may be different for different algorithms. We refer the reader to the complete table available at `http://www.inf.usi.ch/phd/rollini/hvc.html` for more details.

6 Conclusion

We have presented a proof reduction algorithm that is based on an exhaustive set of local transformation rules. Each rule can be applied with respect to a particular local context. In our framework reduction rules (that effectively prune the proof) can be interleaved with a rule that locally perturbates the topology of the proof, in order to create new opportunities for reduction.

We have compared our approach with a previous work with which we share the idea of eliminating redundant pivots from a path from the root to the leaves. In contrast to previous work, our framework can be parameterized with a particular instantiation strategy for the rules, and with a hard limit in the amount of transformations to be carried out.

We implemented both methods plus a hybrid approach and we ran an extensive experimental evaluation over a set of benchmarks from the SMT-LIB. The results show that the hybrid approach yields a higher level of reduction in the proof size.

As future work we would like to investigate the effect of the proof reduction algorithms on some particular application, and in particular in interpolantion-based model checking. Also we plan to derive more efficient and controlled strategies for the application of the rewrite rules and their combination with previous approaches.

References

1. H. Amjad. Compressing Propositional Refutations. In *AVoCS*, pages 7–18, 2006.
2. H. Amjad. Data Compression for Proof Replay. *J. Autom. Reasoning*, 41(3/4), 2008.
3. N. Amla and K. McMillan. Automatic Abstraction Without Counterexamples. In *TACAS*, pages 2–17, 2003.
4. O. Bar-Ilan, O. Fuhrmann, S. Hoory, O. Shacham, and O. Strichman. Linear-Time Reductions of Resolution Proofs. In *HVC*, pages 114–128, 2008.
5. R. Bruttomesso, S. Rollini, N. Sharygina, and A. Tsitovich. Flexible Interpolation with Local Proof Transformations. To appear in ICCAD 2010. Draft available at `http://www.inf.usi.ch/postdoc/bruttomesso/ICCAD2010`.
6. S. Cotton. Two Techniques for Minimizing Resolution Proofs. In *SAT*, pages 306–312, 2010.
7. V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Restructuring Resolution Refutations for Interpolation. Technical report, ETH, 2008.
8. V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant Strength. In *VMCAI*, pages 129–145, 2010.
9. T. Henzinger and K. L. McMillan R. Jhala, R. Majumdar. Abstractions from Proofs. In *POPL*, 2004.
10. R. Jhala and K.L. McMillan. Interpolant-Based Transition Relation Approximation. In *CAV*, pages 39–51, 2005.
11. K. L. McMillan. Interpolation and SAT-Based Model Checking. In *CAV*, pages 1–13, 2003.
12. K. L. McMillan. An Interpolating Theorem Prover. In *TACAS*, pages 16–30, 2004.
13. R.Bruttomesso, E.Pek, N.Sharygina, and A.Tsitovich. The OpenSMT solver. In *TACAS*, 2010.
14. C. Sinz. Compressing Propositional Proofs by Common Subproof Extraction. In *EUROCAST*, pages 547–555, 2007.
15. C. Sinz, A. Kaiser, and W. Kuchlin. Formal Methods for the Validation of Automotive Product Configuration Data. *AI EDAM*, 17(1):75–97, 2003.
16. G. Tseitin. On the Complexity of Proofs in Propositional Logic. *Automation of Reasoning: Classical Papers in Computational Logic 1967-1970*, 2, 1983.
17. Lintao Zhang and Sharad Malik. Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In *DATE*, 2003.