Form Methods Syst Des (2014) 45:1–41
DOI 10.1007/s10703-014-0208-x

# Resolution proof transformation for compression and interpolation

**Simone Fulvio Rollini · Roberto Bruttomesso · Natasha Sharygina · Aliaksei Tsitovich**

Published online: 27 April 2014


**Abstract** Verification methods based on SAT, SMT, and theorem proving often rely on proofs of unsatisfiability as a powerful tool to extract information in order to reduce the overall effort. For example a proof may be traversed to identify a minimal reason that led to unsatisfiability, for computing abstractions, or for deriving Craig interpolants. In this paper we focus on two important aspects that concern efficient handling of proofs of unsatisfiability: compression and manipulation. First of all, since the proof size can be very large in general (exponential in the size of the input problem), it is indeed beneficial to adopt techniques to compress it for further processing. Secondly, proofs can be manipulated as a flexible preprocessing step in preparation for interpolant computation. Both these techniques are implemented in a framework that makes use of local rewriting rules to transform the proofs. We show that a careful use of the rules, combined with existing algorithms, can result in an effective simplification of the original proofs. We have evaluated several heuristics on a wide range of unsatisfiable problems deriving from SAT and SMT test cases.

**Keywords** Resolution system · Craig interpolation · Proof compression · Formal verification

S. F. Rollini (✉) · N. Sharygina
University of Lugano, Lugano, Switzerland
e-mail: simone.rollini@gmail.com; rollinis@usi.ch

N. Sharygina
e-mail: natasha.sharygina@usi.ch

R. Bruttomesso
Atrenta Advanced R&D of Grenoble, Grenoble, France
e-mail: roberto@atrenta.com

A. Tsitovich
Phonak, Stäfa, Switzerland
e-mail: aliaksei.tsitovich@phonak.com

## 1 Introduction

Symbolic verification methods rely on a representation of the state space as a set of formulae, which are manipulated by formal engines such as SAT- and SMT-solvers. For example bounded model checking [9] represents an execution trace leading to a state violating a property as a propositional formula such that the state is reachable if and only if the formula is satisfiable. When the formula is satisfiable, it is possible to infer a counterexample from the model reported by the solver, showing a path that reaches a violating state. When the formula is unsatisfiable, it is instead possible to extract information that better explains the reason why the violating states are unreachable. For instance this can be useful to derive an abstraction of a set of states as it is done in interpolation-based model checking [46] (to abstract the initial states) or IC3 [13] (to derive a minimal set of clauses to put in a frame).

In this paper we describe a set of techniques that allow efficient manipulation of a propositional proof of unsatisfiability, the by-product of an unsatisfiable run of a state-of-the-art solver that may be used to obtain abstractions in the applications mentioned above. In particular we focus on two important aspects: compression of a proof of unsatisfiability, and rewriting to facilitate the computation of interpolants. These approaches are both realized by means of a set of local rewriting rules that enable proof restructuring and compression.

### 1.1 Structure of the paper

The paper is organized as follows. §2 recalls some notions about SAT, SMT and resolution proofs. §3 introduces a proof transformation framework consisting of a set of local rewriting rules and discusses its soundness. §4 addresses the problem of compressing resolution proofs, proposing a collection of algorithms based on the transformation framework. It compares them against existing compression techniques and provides experimental results of running different tools over SMT and SAT benchmarks. §5 presents basic notions about interpolation in first order theories and discusses some limitations of state-of-the-art interpolation algorithms. It then proposes an application of the transformation framework aimed at reordering resolution proofs, in such a way that interpolation is made possible. The approach is demonstrated to be theoretically sound and experiments are provided to show that it is also practically efficient. An algorithm is also provided to reorder resolution steps in a propositional proof to guarantee the generation of interpolants in conjunctive and disjunctive normal form. §6 discusses some of the heuristics adopted in the application of rules by the transformation framework, with reference to §4 and §5. §7 reviews the existing literature on proof manipulation; §8 draws the conclusions.

### 1.2 Improvement over previous own work

The present work builds upon and extends [17] and [56] in a number of ways: (i) it gives a unified and richer description of the Local Transformation Framework and of the set of rewriting rules on which this is based (§3); (ii) it provides a more thorough comparison between the notions of resolution proof trees and DAGs, describing the application and the effect of the rules (§2, §3); (iii) it gives a proof of correctness of the rules and of the SubsumptionPropagation algorithm presented in [17] (§3); (iv) it proposes a new meta-algorithm for proof transformation, TransformAndReconstruct, discussing its soundness and how it can be instantiated to concrete algorithms with different goals (§3); (v) two new compression algorithms, PushdownUnits and StructuralHashing, are proposed and discussed, as well as their combination with the algorithms in [17] and [56] (§4); (vi) a thorough

evaluation of previous and novel algorithms is carried out on a set of purely propositional benchmarks from the literature (§4); (vii) in the context of interpolation we illustrate an application of the Local Transformation Framework to reorder the pivots in a proof so as to guarantee the generation of interpolants in conjunctive normal form (CNF) and DNF (§5); (viii) a description of the heuristics adopted in the application of the rewriting rules has been added, with reference both to compression and to transformation for interpolation (§6).

## 2 Background

The context of this paper is first order logic. We assume countable sets of individual variables $(x, y, z)$, function $(f, g)$ and predicate $(P, Q)$ symbols. A function symbol of 0-arity is called a constant $(a, b, c)$, while a predicate symbol of 0-arity corresponds to a propositional variable $(o, p, q, r)$. A term is built from function symbols and individual variables $(f(c, g(x)))$; an atom is built from predicate symbols and terms $(P(x, f(c)))$. A literal $(s, t)$ is either an atom (having *positive polarity*) or its negation (having negative polarity). A formula $(\phi, \psi)$ is built from atoms and connectives; we are only interested here in quantifier-free formulae. A sentence (or ground formula) is a formula without free variables. A *clause* $C$ is a finite disjunction of literals; a formula in CNF is a finite conjunction of clauses. The empty clause, which represents unsatisfiability, is denoted by $\bot$. We write clauses as lists of literals and sub-clauses, omitting the "$\vee$" symbol, as for instance $p\overline{q}D$ (an overline denotes negation). We use $C_1 \subseteq C_2$ to indicate that $C_1$ *subsumes* $C_2$, that is the set of literals $C_1$ is a subset of the set of literals $C_2$. Also we assume that clauses do not contain duplicated literals or both the occurrence of a literal and its negation. Finally, we use $v(s)$ to denote the variable associated with a literal $s$.

A *SAT-solver* is a decision procedure that solves the propositional satisfiability problem; most successful state-of-the-art solvers rely on variants of the *DPLL* algorithm, as the *conflict-driven clause-learning (CDCL)* [8,45], which are based on the *resolution inference system* [35]. A first order *theory* $\mathcal{T}$ is a collection of sentences; we call $SMT(\mathcal{T})$ the problem of deciding the satisfiability of a formula w.r.t. a theory $\mathcal{T}$. A *theory solver* is an algorithm that decides whether a conjunction of ground literals is satisfiable in $\mathcal{T}$. If the conjunction is unsatisfiable in $\mathcal{T}$, then its negation is valid and is called a $\mathcal{T}$*-lemma*: intuitively, $\mathcal{T}$-lemmata are formulae that encode facts valid in the theory $\mathcal{T}$. An $SMT(\mathcal{T})$*-solver* is a procedure to solve SMT($\mathcal{T}$); in particular, a *lazy* solver integrates a theory solver with a CDCL SAT-solver [59].

### 2.1 The resolution system

The *resolution system* is an inference system based on a single inference rule, called *resolution rule*:

$$\frac{p\,D \qquad \overline{p}E}{DE}\ p$$

Clauses $pD$ and $\overline{p}E$ are the *antecedents*, $DE$ is the *resolvent* and $p$ is the *pivot* variable. We also represent a *resolution step* (an application of the resolution rule) as $DE = Res_p(pD, \overline{p}E)$.

SAT- and SMT-solvers can be instrumented to generate, for unsatisfiable formulae, a certificate of unsatisfiability in the form of a *proof of unsatisfiability* or *refutation*. It is straightforward to instruct a state-of-the-art CDCL solver to return proofs: a resolution proof, in particular, can be derived by logging the inference steps performed during conflict analysis [69].

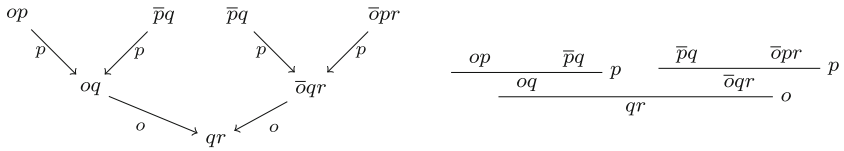



**Fig. 1** Resolution proof tree

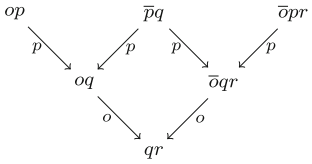



**Fig. 2** Resolution proof DAG

Throughout the paper we shall use the notions of *resolution proof tree* and *resolution proof DAG*.

**Definition 1** (*Resolution Proof Tree*) A *resolution proof tree* of a clause $C$ from a set of clauses $\mathbb{C}$ is a tree such that:

1. Each node $n$ is labeled by a clause $C(n)$.
2. If $n$ is a leaf, $C(n) \in \mathbb{C}$.
3. The root is a node $n$ s.t. $C(n) = C$.
4. An inner node $n$ has pivot $piv(n)$ and exactly two parents $n^+, n^-$ s.t. $C(n) = Res_{piv(n)}(C(n^+), C(n^-)))$. $C(n^+)$ and $C(n^-)$ respectively contain the positive and negative occurrence of the pivot.
5. Each non-root node has exactly one child.

In the following, we equivalently use a graph-based representation (left) or an inference rule-based representation (right) (Fig. 1).

In real-world applications proofs are rarely generated or stored as trees; for instance proofs produced by CDCL solvers are represented as directed acyclic graph (DAGs). We therefore introduce the following notion of resolution proof, which is more suitable for describing the graph-based transformation algorithms illustrated in this paper.

**Definition 2** (*Resolution Proof DAG*) A *resolution proof DAG* of a clause $C$ from a set of clauses $\mathbb{C}$ is a directed acyclic graph such that:

1.–4. hold as in Def. 1.
5. Each non-root node has one or more children.

Resolution proof DAGs extend the notion of resolution proof trees by allowing a node to participate as antecedent in multiple resolution steps (Fig. 2).

We identify a node $n$ with its clause $C(n)$ whenever convenient; in general, different nodes can be labeled by the same clause, that is $C(n_i) = C(n_j)$ for $i \neq j$. A proof $P$ is a *refutation* if $C = \bot$. A *subproof* $P'$, with *subroot* $n$, of a proof $P$ is the subtree that derives $C(n)$ from a subset of clauses that label leaves of $P$; when referring to $P$ and its root compared to $P'$, we call $P$ *global proof* and its root *global root*.

It is always possible to turn a resolution proof tree into a resolution proof DAG, by merging two or more nodes labeled by a same clause into a single node, which inherits the children of the merged nodes. On the other hand, a resolution proof DAG can be "unrolled" into a

resolution proof tree, possibly at exponential cost: it is in fact sufficient to traverse the DAG bottom–up, duplicating nodes with multiple children so that each node is left with at most one child.

Similarly to [5], we distinguish between a *legal* and an *illegal* proof; an illegal proof is a proof which has undergone transformations in such a way that some clauses might not be the resolvents of their antecedents anymore. In this paper however an illegal proof represents an intermediate transformation step in an algorithm, and the proof can always be *reconstructed* into a legal one, as explained in the next sections.

In the following, we will consider refutations as obtained by means of modern CDCL SAT-solvers and lazy SMT-solvers, involving both propositional and theory atoms. Whenever the theory content is not relevant to the problem at hand, it is convenient to represent each theory atom with a new propositional variable called its *propositional abstraction*: for example an atom $x + y < 1$ will be represented by a certain variable $q$.

### 2.2 Resolution proofs in verification

Resolution proofs find application in many verification techniques. For instance, Amla and McMillan's [4] 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 specifications of inconsistency in various industrial applications (e.g., product configuration or declarative modeling [60,62]). In the context of proof-carrying code [50] a system can verify a property about an application exploiting a proof provided with the application executable code. SAT-solvers and SMT-solvers can be integrated into interactive theorem provers as automated engines to produce proofs, that can be later replayed and verified within the provers [3,29,66]. An unsatisfiable core, that is an inconsistent subset of clauses, can be extracted from a proof, to be exploited for example during the refinement phase in model checking [4,37]. Another noteworthy application is in the framework of interpolation-based model checking, where interpolants are generated from proofs based on their structure and content [39,46–48].

## 3 The Local Transformation Framework

This section introduces a proof transformation framework based on local rewriting rules. We start by assuming a resolution proof tree, and then extend the discussion to resolution proof DAGs. All results related to proofs hold in particular for refutations.

The framework is built on a set of rewriting rules that transform a subproof with root $C$ into one whose subroot $C'$ is logically equivalent or stronger than $C$ (that is, $C' \implies C$). Each rewriting rule is defined to match a particular *context*, identified by two consecutive resolution steps (see Fig. 3). A context involves two pivots $p$ and $q$ and five clauses $C_1, C_2, C_4, C_3, C$; we call $C$ the *context root*; the subproof rooted in $C$ is the *context subproof*. Clearly $p$ is contained in $C_1$ and $C_2$ (with opposite polarity), and $q$ is contained in $C_4$ and $C_3$ (again with opposite polarity); $q$ must be contained in $C_1 \cup C_2$.

A clause $C$ might be the root of two different contexts, depending on whether $C_1$ and $C_2$ are taken as the antecedents of either of the two antecedents of $C$; in that case, to distinguish among them we talk about *left* and *right context*.

Figure 4 shows a set of proof transformation rules. Each rule is associated with a unique context, and, conversely, each context can be mapped to at least one rule (i.e., the set of rules is exhaustive, modulo symmetry, for every possible context). A first property that characterizes

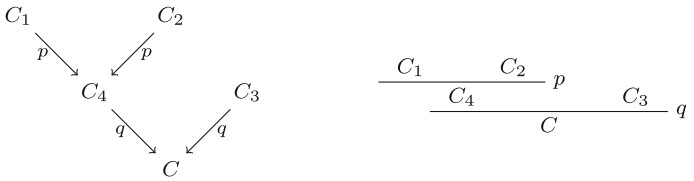



**Fig. 3** Rule context

the set of rules is *locality*: only the limited information represented by a context is in fact needed to determine which rule is applicable. A second property is *strengthening*: the rules either keep the context root unchanged or turn it into a logically stronger formula.

The classification of rules into $S$ (swapping) and $R$ (reducing) depends on the effect of the rules on the context rooted in $C$: $S1$ and $S2$ swap the two resolution steps in the context without modifying $C$, while $R1$, $R2$, $R2'$ and $R3$ replace $C$ with a new $C'$ such that $C' \subseteq C$; in other words, the $R$ rules generate subproofs with stronger roots.

The influence of the $S$ rules does not extend beyond the context where they are applied, while that of the $R$ rules possibly propagates down to the global root. The $R$ rules essentially simplify the proof and their effect cannot be undone, while an application of an $S$ rule can be reversed. In particular, the effect of rule $S2$ can be canceled out simply by means of another application of the same $S2$. $S1$ has $S1'$ as its inverse (notice the direction of the arrow); $S1'$ is actually a derived rule, since it corresponds to the sequential application of $S2$ and $R2$.

The rules $R2$ and $R2'$ are associated with the same context; they respectively behave as $S2$ (with an additional simplification of the root) and $R1$. The decision whether to apply either rule depends on the overall goal of the transformation. Note that the application of rule $R2$ to a context turns it into a new context which matches rule $S1$.

### 3.1 Extension to resolution proof DAGs

If the proof to be transformed is a DAG rather than a tree, some constraints are necessary on the application of the rules.

Consider rules $S1$, $S1'$, $S2$, $R2$, and suppose clause $C_4$ is involved in more than one resolution step, having thus at least another resolvent $C_5$ besides $C$. If $C_4$ is modified by a rule, it is not guaranteed that the correctness of the resolution step having $C_5$ as resolvent (and in turn of the resolution steps on the path from $C_5$ to the global root) is preserved. This problem does not concern clauses $C_1$, $C_2$, $C_3$ and the subproofs rooted in them, which are not changed by any rule.

A simple solution consists in creating a copy of $C_4$, to which all resolvents of $C_4$ besides $C$ are assigned, so that $C_4$ is left with exactly one resolvent; at that point any modification to $C_4$ will affect only the context rooted in $C$. Since duplications increase the size of the proof, they should be carried out with moderation (see §6).

A more efficient alternative exists in case of rules $R1$, $R2'$, $R3$, where $C_4$ is detached by the context rooted in $C$ and loses $C$ as resolvent, but keeps the other resolvents (if any). The effect of the transformation rules is shown in Fig. 5: the presence of additional resolvents for $C_4$ is denoted by a dotted arrow.

### 3.2 Soundness of the Local Transformation Framework

In this section we first prove that the rewriting rules preserve the legality of the subproofs rooted in the contexts where the rules are applied; then we discuss how the rules affect the global proof and what steps must be taken to maintain it legal.

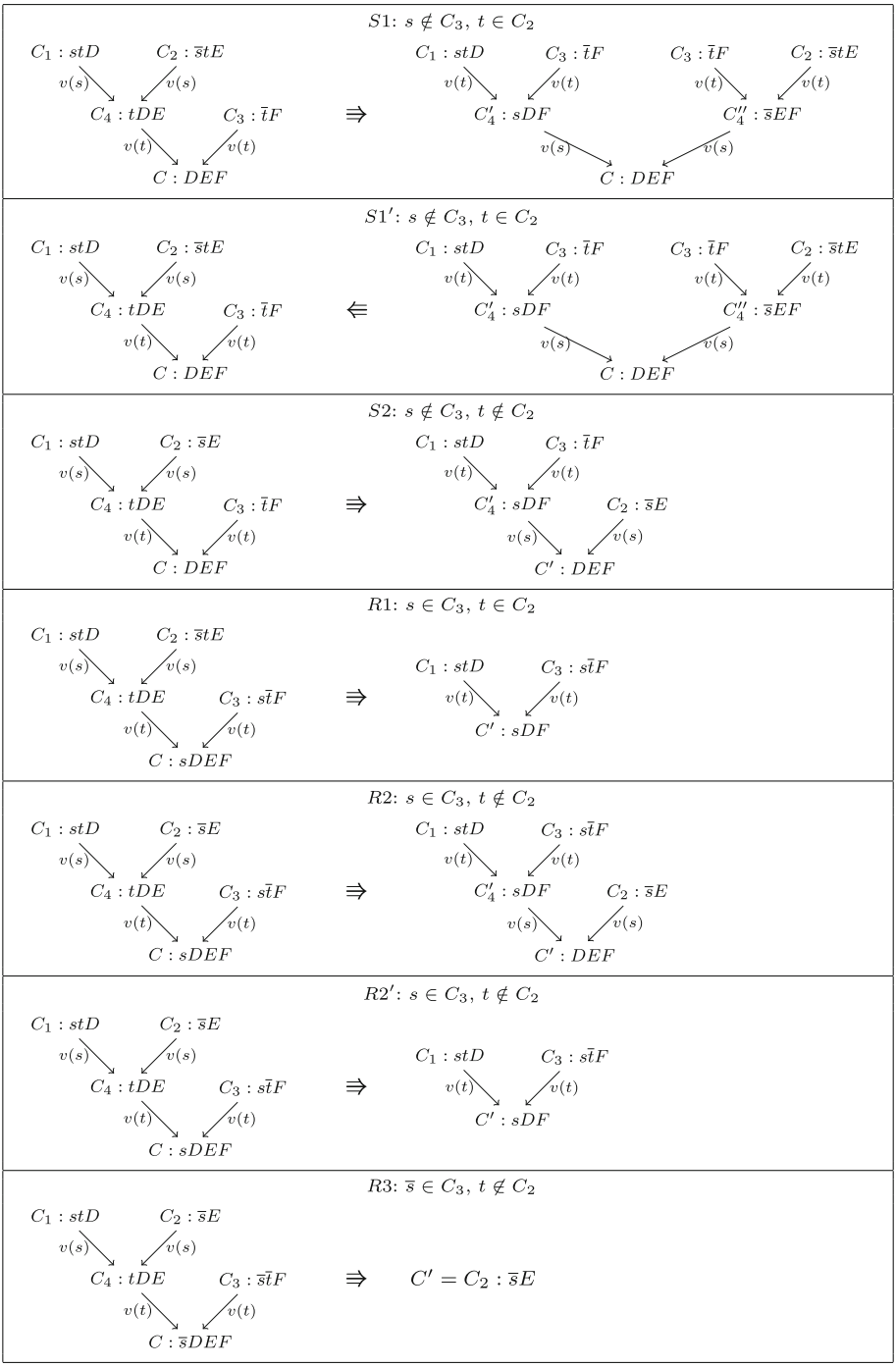



**Fig. 4** Local transformation rules for resolution proof trees

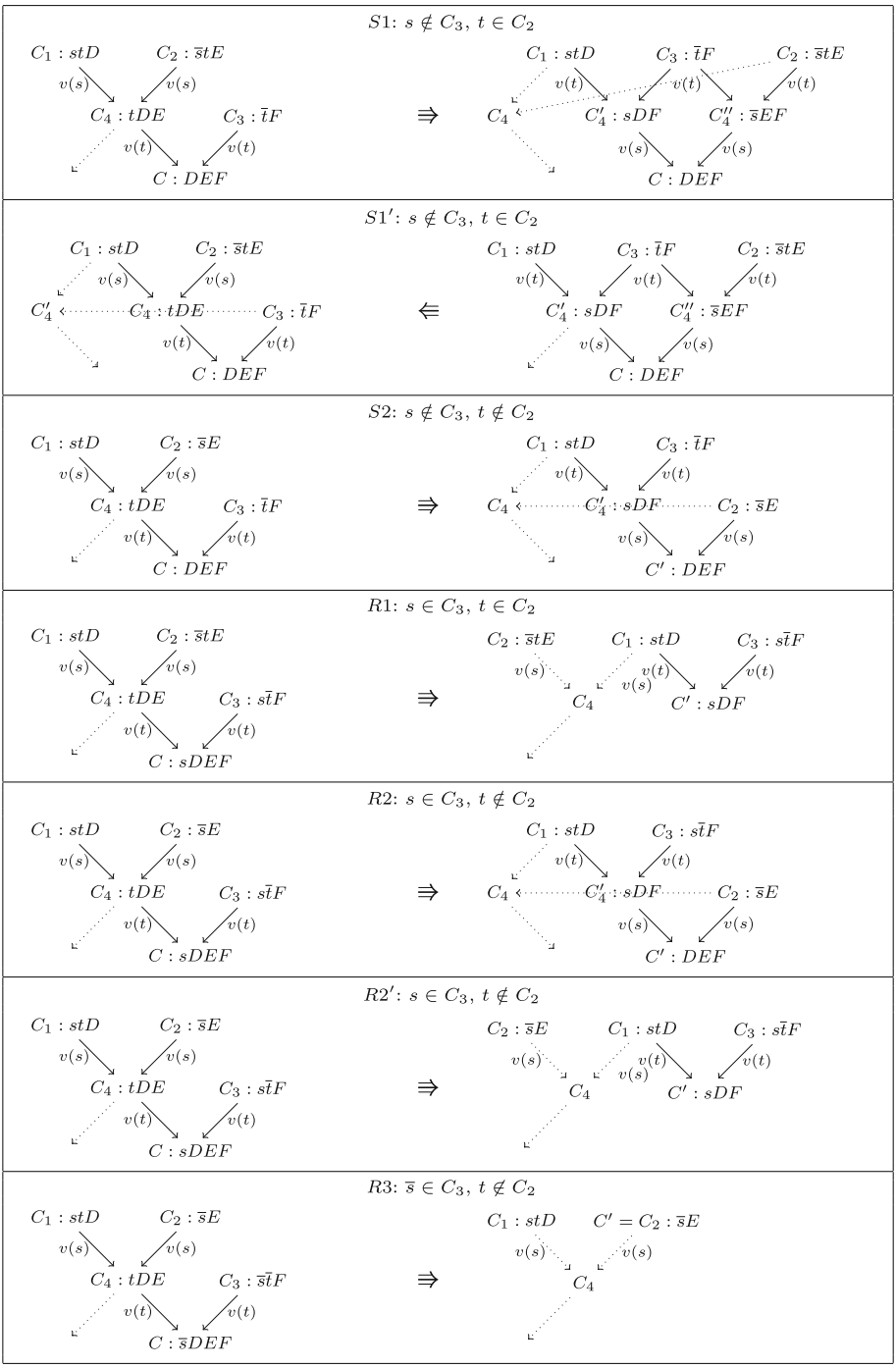



**Fig. 5** Local transformation rules for resolution proof DAGs

### 3.2.1 Effect on a context

Based on the following observations, we claim that after a single application of a rule to a context with root $C$, the legal subproof rooted in $C$ is replaced by a legal subproof rooted in $C' \subseteq C$.

Refer to Fig. 5. No additional subproofs are introduced by the rules and no modifications are brought to the subproofs rooted in $C_1, C_2, C_3$, which are simply recombined or detached from the context. As for the $S$ rules, $C_4$ is either replaced by the resolvent of $C_1, C_3$ ($S2$) or by the resolvent of the resolvents of $C_1, C_3$ and $C_3, C_2$ ($S1$, where a new clause $C_4'' = Res_{v(s)}(C_2, C_3)$ is also introduced). Note that in both cases $C$ is not modified. The $R$ rules instead yield a more substantial change in the form of a stronger context root $C' \subseteq C$:

- In $R1$ and $R2'$, the subproofs with root $C_1$ and $C_3$ are combined to obtain a subproof with root $sDF \subseteq sDEF$.
- $R2$ has a swap effect similar to $S2$, but replaces the root $sDEF$ with $DEF$, removing a single literal.
- In $R3$, the whole subproof is substituted by the subproof rooted in $C_2 = \overline{s}E$, which subsumes $C = \overline{s}DEF$.

All the above transformations involve updating the relevant clauses by means of sound applications of the resolution rule.

### 3.2.2 Effect on the global proof

The application of a rule to a context yields a legal subproof rooted in a clause $C' \subseteq C$; however, the global proof could turn into an illegal one. In fact, the deletion of literals from $C$ affects the sequence of resolution steps that extends from $C$ to the global root: some of these steps might become superfluous, because they resolve upon a variable which was introduced by $C$ (but does not appear in $C'$), and they should be appropriately removed. In the same way, the elimination of a resolution step could itself lead to the disappearance of more literal occurrences, leading to a chain reaction.

The following Algorithm 1, *SubsumptionPropagation*, has the purpose of propagating the effect of the replacement of $C$ by $C' \subseteq C$ along the path leading from $C$ to the global root.

The algorithm restructures the proof in a top–down manner analyzing the sequence of resolution steps to ensure their correctness while propagating the effect of the initial subsumption. We prove that, after an execution of SubsumptionPropagation following the application of an $R$ rule to a legal proof, the result is still a legal proof.

The idea at the base of the algorithm reflects the mechanisms of the restructuring procedures first proposed in [5,28]:

1. It determines the effect range of the substitution of $C$ by $C'$, which corresponds to the set of nodes reachable from the node labeled by $C'$.
2. It analyzes, one by one, all reachable nodes; it is necessary that the antecedents of a node $n$ have already been visited (and possibly modified), in order to guarantee a correct propagation of the modifications to $n$.
3. Due to the potential vanishing of literals from clauses, it might happen that in some resolution step the pivot is not present in both antecedents anymore; if that is the case, the resolution step is deleted, by replacing the resolvent with the antecedent devoid of the pivot (if the pivot is missing in both antecedents, either of them is arbitrarily chosen), otherwise, the resolution step is kept and the resolvent clause updated. At the graph level, $n$ is substituted by $n^+$ or $n^-$, assigning the children of $n$ (if any) to it.

**Algorithm 1:** SubsumptionPropagation.

**Input**: A legal proof modified by an $R$ rule
**Output**: A legal proof
**Data**: $W$: set of nodes reachable from $C'$, $V$: set of visited nodes

```
1  begin
2      V ← ∅
3      Determine W, e.g. through a visit from C'
4      while W \ V ≠ ∅ do
5          Choose n ∈ W \ V such that:
6          (n+ ∈ W or n+ ∉ W) and (n− ∈ W or n− ∉ W)
7          V ← V ∪ {n}
8          p ← piv(n)
9          if p ∈ C(n+) and p̄ ∈ C(n−) then
10             C(n) ← Res_p(C(n+), C(n−))
11         else if p ∉ C(n+) and p̄ ∈ C(n−) then
12             Substitute n with n+
13         else if p ∈ C(n+) and p̄ ∉ C(n−) then
14             Substitute n with n−
15         else if p ∉ C(n+) and p̄ ∉ C(n−) then
16             Heuristically choose a parent, replace n with it
17     end
18 end
```

**Theorem 1** *Assume a legal proof $P$. The application of an $R$ rule, followed by an execution of SubsumptionPropagation, yields a legal proof $P'$, whose new global root subsumes the previous one.*

*Proof (by structural induction)*

***Base case*** Assume an $R$ rule is applied to a context rooted in a clause $C$; $C$ is replaced by $C' \subseteq C$ and the subproof rooted in $C'$ is legal, as previously shown. The subproofs rooted in the clauses of nodes not reachable from $C$ are not affected and thus remain legal.

***Inductive step*** All nodes reachable from $C$ are visited; in particular, a node $n$ is visited after its reachable parents. By inductive hypothesis $C'(n^+) \subseteq C(n^+)$, $C'(n^-) \subseteq C(n^-)$ and the subproofs rooted in $C'(n^+)$ and $C'(n^-)$ are legal. We show that, after visiting $n$, $C'(n) \subseteq C(n)$ and the subproof rooted in $C'(n)$ is legal. Let $p = piv(n)$.

We have three possibilities:

- Case 1: the pivot still appears both in $C'(n^+)$ and in $C'(n^-)$; $C'(n) = Res_p(C'(n^+), C'(n^-))$, thus $C'(n) \subseteq C(n)$.
- Case 2: the pivot is present only in one antecedent, let us say $C'(n^+)$; the subproof rooted in $C(n)$ is replaced by the one rooted in $C'(n^-)$ (legal by hypothesis). But $C'(n) = C'(n^-) \subseteq C(n)$ since $C'(n^-)$ does not contain the pivot.
- Case 3: the pivot is not present in either antecedent. Same reasoning as for Case 2, but arbitrarily choosing an antecedent for the substitution.

In all three cases the subproof rooted in $C'(n)$ is legal and $C'(n) \subseteq C(n)$. □

Figure 6 shows the effect of $R2$ and the subsequent application of SubsumptionPropagation on a small proof.

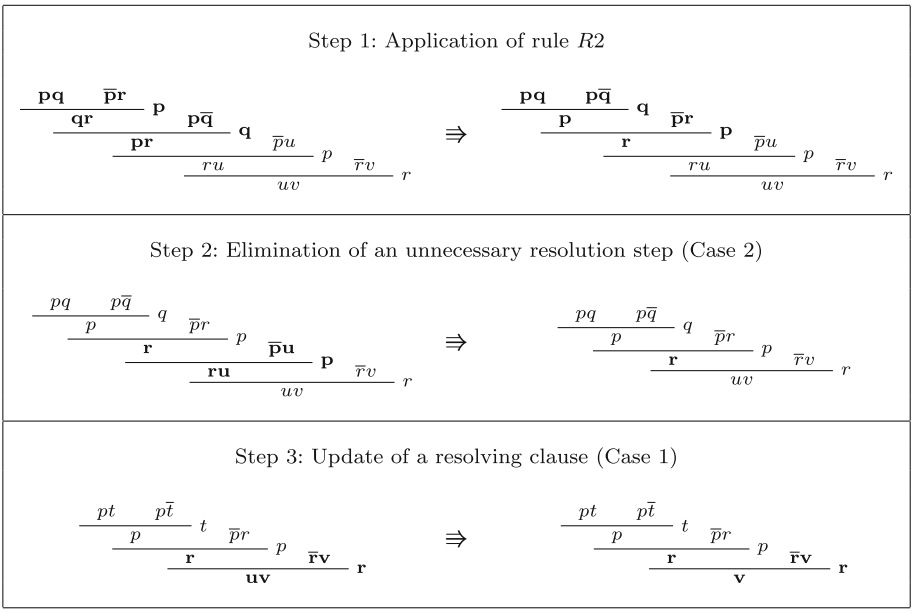



**Fig. 6** Example of rule application and subsumption propagation

### 3.3 A transformation meta-algorithm

The Local Transformation Framework defined by our rules leaves to the user the flexibility of choosing a particular *strategy* and a *termination criterion* for their application.

Whenever a sizeable amount of rules has to be applied, rather than running Subsumption-Propagation multiple times, it is more efficient to combine the application of all rules and the propagation of the modifications into a single traversal of the proof.

Algorithm 2, *TransformAndReconstruct*, illustrates this approach. At first it performs a topological sorting of the proof (line 2), in order to ensure that each node is visited after its parents. Then it analyzes one node at a time, checking if the corresponding resolution step is still sound (line 6). If the resolution step is sound, 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 (line 9). If the resolution step is not sound and either antecedent does not contain the pivot (lines 11, 13, 15), then the resolution step is removed by replacing the resolvent with that antecedent (which, missing the pivot, subsumes the resolvent); at the graph level, $n$ is substituted by $n^+$ or $n^-$.

Note that the antecedent not responsible for the substitution might have lost all its resolvents and thus does not contribute to the proof anymore; in that case it is pruned away, together with the portion of the subproof rooted in it which has become detached from the global proof.

A key point of the algorithm is the call to *ApplyRule(left context, right context)*: this method heuristically chooses at most one context (possibly none) rooted in $n$ and applies the corresponding rule. The instantiation of ApplyRule with different procedures yields concrete algorithms suitable for particular applications, as illustrated in the next sections.

Based on the above observations and on Theorem 1, we have the following result:

**Theorem 2** *TransformAndReconstruct outputs a legal proof.*

**Algorithm 2:** TransformAndReconstruct.

```
Input: A legal proof, an instance of ApplyRule
Output: A legal proof
Data: TS: nodes topological sorting vector
1  begin
2      TS ← topological_sorting_top_down(proof)
3      foreach n ∈ TS do
4          if n is not a leaf then
5              p ← piv(n)
6              if p̄ ∈ C(n⁻) and p ∈ C(n⁺) then
7                  C(n) ← Res_p(C(n⁻), C(n⁺))
8                  Determine left context lc of n, if any
9                  Determine right context rc of n, if any
10                 ApplyRule(rc, lc)
11             else if p̄ ∉ C(n⁻) and p ∈ C(n⁺) then
12                 Substitute n with n⁻
13             else if p̄ ∈ C(n⁻) and p ∉ C(n⁺) then
14                 Substitute n with n⁺
15             else if p̄ ∉ C(n⁻) and p ∉ C(n⁺) then
16                 Heuristically choose a parent, substitute n with it
17     end
18 end
```

## 4 Proof compression

Resolution proofs, as generated by modern solvers, find application in many verification techniques. In most cases, the size of the proofs affects the efficiency of the methods in which they are used. It is known that the size of a resolution proof can grow exponentially with respect to the size of the input formula: even when proofs are representable in a manageable memory space, it might be crucial for efficiency to reduce or compress them as much as possible. Several compression technique have been developed and can be found in literature, ranging from memoization of common subproofs to partial regularization [2,3,5,21,28,30,61]; however, since the problem of finding a minimum proof is NP-hard, it is still an open challenge to design heuristics capable of obtaining good reduction in practical situations.

This section discusses algorithms aimed at compressing proofs. We identify two kinds of *redundancies* in resolution proofs and present a set of post-processing techniques aimed at removing them; the techniques are independent from the way the refutation is produced and can be applied to an arbitrary resolution proof of unsatisfiability. We also illustrate how to combine these algorithms in an effective manner, and show the results of experimenting on a collection of SAT and SMT benchmarks.

We do not address directly the problem of core minimization, that is nonetheless achieved as a side effect of proof reduction. A rich literature exists on techniques aimed at obtaining a minimum (a $\Sigma_2$-complete problem), minimal ($D^P$-complete), or small unsatisfiable core, that is a subset of the initial set of clauses that is still unsatisfiable [15,19,26,36,41,44,49,52,68].

### 4.1 Proof redundancies

This paper focuses on two particular kinds of redundancies in resolution proofs.

The first one stems from the observation that, along each path from a leaf to the root, it is unnecessary to resolve upon a certain pivot more than once. The proof can be simplified, for example by keeping (for a given variable and a path) only the resolution step closest to the root, while cutting the others away. In the literature, a proof such that each variable is used as a pivot at most once along each path from a leaf to the root is said to be *regular* [64].

The second kind of redundancy is related to the content of a proof. It might be the case that there exist multiple nodes associated with equal clauses; such nodes can be merged, keeping only one pair of parents and grouping together all the children. In particular, we call a proof *compact* if $C(n_i) = C(n_j) \implies i = j$ for any $i, j$, that is, different nodes are labeled by different clauses.

## 4.2 Proof regularity

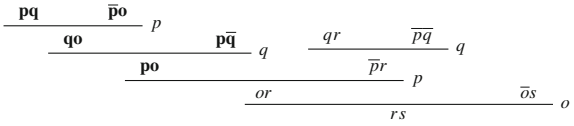
In this section we discuss how to make a proof (partially) regular. We show how to employ Algorithm 2 for this purpose and present two algorithms explicitly devised for regularization, namely RecyclePivots [5] and its refinement RecyclePivotsWithIntersection [30]. We illustrate them individually and explain how they can be combined to obtain more powerful algorithms.

### *4.2.1 Regularization in the Local Transformation Framework*

The $R$ rules are, as a matter of fact, a means to perform a "local" regularization; they are applied to contexts where a resolution step on a pivot $v(s)$ is immediately followed by a reintroduction of the pivot with positive ($R1$, $R2$, $R'2$) or negative ($R3$) polarity (see Fig. 5).

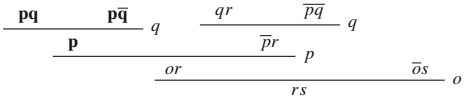
Resolving on $v(s)$ is redundant, since the newly introduced occurrence of the pivot will be later resolved upon along the path to the global root; the $R$ rules have the effect of simplifying the context, possibly pruning subproofs which do not contribute anymore to the global proof. Moreover, the rules replace the root of a context with a stronger one, which allows to achieve further compression as shown below.

Consider, for example, the following proof:



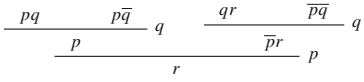

(1)

The highlighted context can be reduced via an application of $R2$ as follows:




(2)

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

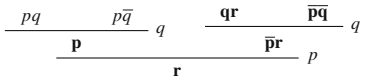



(3)

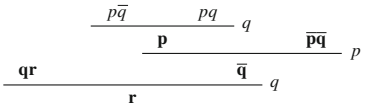
At this stage no other $R$ rule can be directly applied to the proof.

Rule $S2$ does not perform any simplification 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 $R$ rules to be applied.

Consider again our running example. $S2$ can be applied as follows:

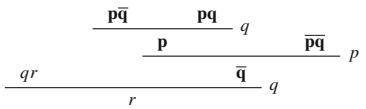



(4)

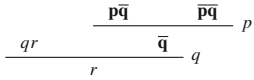



(5)

$S2$ has now exposed a new redundancy involving the variable $q$. The proof can be readily simplified by means of an application of $R2'$:




(6)




(7)

As discussed in §3.3, the rewriting framework defined by our rules allows the flexibility of choosing a strategy and a termination criterion for their application.

A simple strategy is to eagerly apply the $R$ rules until possible, shuffle the proof by means of $S2$ with the purpose of disclosing other redundancies, and then apply the $R$ rules again, in an iterative fashion. However there is usually a very large number of contexts where $S2$ could be applied, and it is computationally expensive to predict whether one or a chain of $S2$ applications would eventually lead to the creation of contexts for an $R$ rule.

For efficiency reasons, we rely on the meta-algorithm described in Algorithm 2, for a particular instantiation of the ApplyRule method. Algorithm 2 does a single traversal of the proof, performing shuffling and compression; it is run multiple times, setting a number of traversals to perform and a timeout as termination criteria (whichever is reached first). The resulting regularization procedure is *ReduceAndExpose*, listed as Algorithm 3.

**Algorithm 3:** ReduceAndExpose.

**Input**: A legal proof; *timelimit*: timeout; *numtrav*: number of transformation traversals; an instantiation of *ApplyRule*
**Output**: A legal proof

```
1 begin
2     for i=1 to numtrav do
3         TransformAndReconstruct(ApplyRule)
4         if timelimit is reached then
5             break
6     end
7 end
```

### 4.2.2 The RecyclePivots approach

The RecyclePivots algorithm was introduced in [5] as a linear-time technique to perform a partial regularization of resolution proofs.

RecyclePivots is based on analyzing the paths of a proof, focusing on the pivots involved in the resolution steps; if a pivot is resolved upon more than once on a path (which implies that the pivot variable is introduced and then removed multiple times), the resolution step closest to the root is kept, while the others are simplified away.

We illustrate this approach by means of an example. Consider the leftmost path of proof (1). Variable $p$ is used twice as pivot. The topmost resolution step is redundant as it resolves upon $p$, which is reintroduced in a subsequent step (curly brackets denote the set $RL$ of removable literals, see later).

$$\dfrac{\dfrac{\dfrac{\mathbf{pq} \qquad \mathbf{\overline{p}o}}{\mathbf{qo}\ \{\overline{p},\overline{q}\}} p \qquad \mathbf{p\overline{q}}}{\mathbf{po}\ \{\overline{p}\}} q \qquad \dfrac{qo \qquad \overline{pq}}{\overline{p}o} q}{o} p \tag{8}$$

Regularization can be achieved by eliminating the topmost resolution step and by adjusting the proof accordingly. The resulting proof is shown below.

$$\dfrac{\dfrac{\mathbf{pq} \qquad \mathbf{p\overline{q}}}{\mathbf{p}} q \qquad \dfrac{qo \qquad \overline{pq}}{\overline{p}o} q}{o} p \tag{9}$$

Algorithm 4 shows the recursive version of RecyclePivots (RP in the following). It is based on a depth-first visit of the proof, from the root to the leaves. It starts from the global root, having as input a set of *removable literals RL* (initially empty). The removable literals are essentially the (partial) collection of pivot literals encountered during the bottom–up exploration of a path. If the pivot variable of a resolution step under consideration is in $RL$ (lines 15 and 18), then the resolution step is redundant and one of the antecedents may be removed from the proof. The resulting proof is illegal and has to be reconstructed into a legal one, which can be done in linear time, as shown in [5].

Note that in the case of resolution proof trees, the outcome of the algorithm is a regular proof. For arbitrary resolution proof DAGs the algorithm is executed in a limited form (when nodes with multiple children are detected) precisely by resetting $RL$ (line 10); therefore the result is not necessarily a regular proof.

### 4.2.3 RecyclePivotsWithIntersection

The aforementioned limitation is due to the same circumstance that restricts the application of rules in the Local Transformation Framework, as discussed in §3.1. The set of removable literals of a node is computed for a particular path from the root to the node (which is enough in presence of proof trees), but does not take into account the existence of other possible paths to that node. Thus, suppose a node $n$ with pivot $p$ is replaced by one of its parents (let us say $n^+$) during the reconstruction phase, and $C(n^+) \not\subseteq C(n)$; then, it might happen that some of the literals in $C(n^+) \setminus C(n)$ are not resolved upon along *all* paths from $n$ to the root, and are thus propagated to the root, making the proof illegal.

In order to address this issue, the authors of [30] extend RP by proposing RecyclePivotsWithIntersection (RPI), an iterative version of which is illustrated in Algorithm 5. RPI

**Algorithm 4:** 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 child then
10                 RL ← ∅
11             p ← piv(n)
12             if p ∉ RL and p̄ ∉ RL then
13                 RecyclePivots(n+,RL ∪ {p̄})
14                 RecyclePivots(n−,RL ∪ {p})
15             else if p ∈ RL then
16                 n+ ← null
17                 RecyclePivots(n−,RL)
18             else if p̄ ∈ RL then
19                 n− ← null
20                 RecyclePivots(n+,RL)
21 end
```

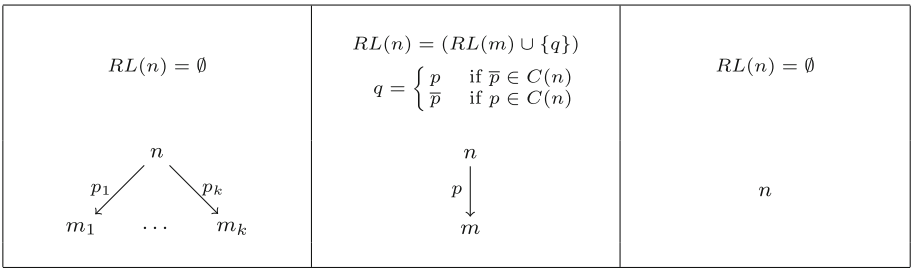



**Fig. 7** Computation of $RL$ in RecyclePivots

refines RP by keeping track for each node $n$ of the set of pivot literals $RL(n)$ which get resolved upon along *all* paths from $n$ to the root.

The computation of $RL$ in the two approaches is represented in Figs. 7 and 8. RP and RPI behave in the same way whenever a node $n$ has only one child. In case $n$ has no children, i.e., it is the root, RPI takes into account the possibility for the root to be an arbitrary clause (rather than only $\bot$, as in refutations) and sets $RL$ to include all variables of $C(n)$; it is equivalent to having a path from $n$ to $\bot$ where all variables of $C(n)$ are resolved upon. The major difference between RP and RPI is in the way a node $n$ with multiple children is handled: RP sets $RL(n)$ to $\emptyset$, while RPI sets $RL(n)$ to the intersection $\bigcap(RL(m_i) \cup q_i)$ of the removable literals sets of its children, augmented with the pivots of the resolution steps of which the children are resolvents.

RPI starts in Algorithm 5 by computing a topological sorting of the nodes (line 2), from the root to the leaves. $RL(root)$ is computed as the set of literals in the root clause; for any other node $n$, $RL(n)$ is initialized and then iteratively refined each time one of its children is visited (lines 10–18). Similarly to RecyclePivots, whenever visiting an inner node $n$, if

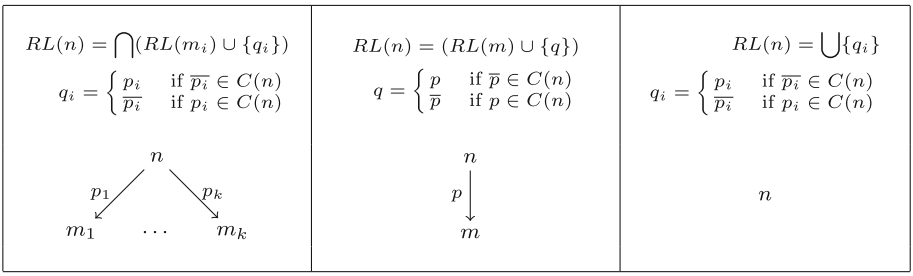



**Fig. 8** Computation of $RL$ in RecyclePivotsWithIntersection

**Algorithm 5:** RecyclePivotsWithIntersection.

**Input**: A legal proof
**Input**: A proof to be reconstructed
**Data**: $TS$: nodes topological sorting vector; $RL$: vector of sets of removable literals;

```
1  begin
2      TS ← topological_sorting_bottom_up(proof)
3      foreach n ∈ TS do
4          if n is not a leaf then
5              if n is the root then
6                  RL(n) ← {p̄_i}_{p_i ∈ C(n)}
7              else
8                  p ← piv(n)
9                  if p ∈ RL(n) then
10                     n⁺ ← null
11                     if n⁻ not seen yet then
12                         RL(n⁻) ← RL(n)
13                         Mark n⁻ as seen
14                     else RL(n⁻) ← RL(n⁻) ∩ RL(n)
15                 else if p̄ ∈ RL(n) then
16                     n⁻ ← null
17                     if n⁺ not seen yet then
18                         RL(n⁺) ← RL(n)
19                         Mark n⁺ as seen
20                     else RL(n⁺) ← RL(n⁺) ∩ RL(n)
21                 else if p ∉ RL(n) and p̄ ∉ RL(n) then
22                     if n⁻ not seen yet then
23                         RL(n⁻) ← (RL(n) ∪ {p})
24                         Mark n⁻ as seen
25                     else RL(n⁻) ← RL(n⁻) ∩ (RL(n) ∪ {p})
26                     if n⁺ not seen yet then
27                         RL(n⁺) ← (RL(n) ∪ {p̄})
28                         Mark n⁺ as seen
29                     else RL(n⁺) ← RL(n⁺) ∩ (RL(n) ∪ {p̄})
30     end
31 end
```

$piv(n)$ appears in $RL(n)$ then the resolution step is redundant and can be simplified away (lines 7–12); in that case, $RL(n)$ is propagated to a parent of $n$ without the addition of $piv(n)$.

Figure 9 shows the effect of RPI on a small proof where RP cannot achieve any compression: RP sets $RL(qr) = \emptyset$ since $qr$ has two children, while RPI sets $RL(qr) = \{\overline{r}, \overline{p}, \overline{q}\}$

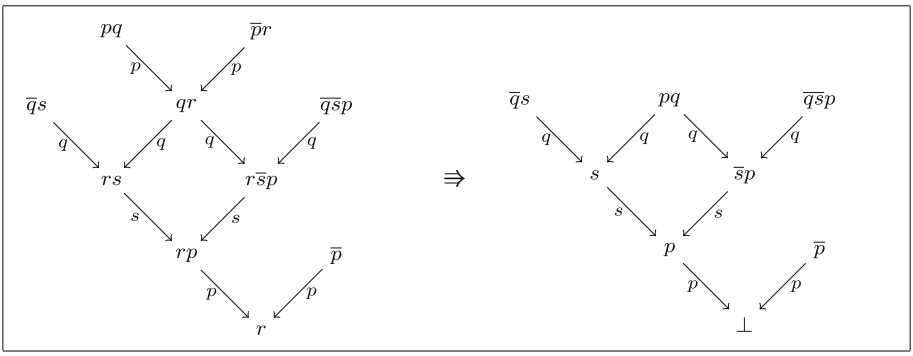



**Fig. 9** Compression of a proof by means of RecyclePivotsWithIntersection

and consequently simplifies the uppermost resolution step, since it is able to detect that $p$ is resolved upon along both paths from $qr$ to the root.

### *4.2.4 RecyclePivots and the Local Transformation Framework*

RecyclePivots (as well as its refinement RecyclePivotsWithIntersection) and ReduceAndExpose both aim at compressing a proof by identifying and removing pivot redundancies along paths from the root to the leaves. The main difference between the two approaches is that RecyclePivots operates on a *global perspective* without changing the topology of the proof (i.e., no shuffling), while ReduceAndExpose operates on *local contexts* and allows the topology to change. Both approaches have advantages and disadvantages.

Operating on a global perspective without modifying the topology allows a one-pass visit and compression of the proof. Maintaining a fixed topology, however, may prevent the disclosure of hidden redundancies. For instance the application of RecyclePivots to the example of §4.2.1 would have stopped to step (3), since no more redundant pivots can be found along a path (the proof is regular). The local contexts instead have to be gathered and considered multiple times. On the other hand, the ability of ReduceAndExpose to change the topology allows more redundancies to be exposed.

Another advantage of RecyclePivots is that it can eliminate redundancies that are separated by many resolution steps. The $R$ rewriting rules instead are applicable only when there is a reintroduction of a certain variable immediately after a resolution step upon it. Such configurations, when not present in the proof, can be produced by means of applications of the $S2$ rule.

The ability of the Local Transformation Framework to disclose redundancies and the effectiveness of RecyclePivots at removing them can be combined in a simple hybrid approach, shown in Algorithm 6.

The algorithm takes as input an overall time limit, a number of *global iterations* and a number of transformation traversals for ReduceAndExpose. The time limit and the amount of global iterations determine the execution time available to ReduceAndExpose during each iteration. ReduceAndExpose and RecyclePivots are run one after the other by Algorithm 6, alternately modifying the topology to expose redundancies and simplifying them away.

A similar, but more efficient algorithm can be obtained by simply replacing the call to RecyclePivots with a call to RecyclePivotsWithIntersection.

**Algorithm 6:** RP + RE.

**Input**: A legal proof; *numloop*: number of global iterations; *numtrav*: number of transformation traversals for each global iteration; *timelimit*: timeout; an instantiation of *ApplyRule*
**Output**: A legal proof

```
1 begin
2     timeslot = timelimit/numloop
3     for i=1 to numloop do
4         RecyclePivots(root,∅)
5         // RPtime is the time taken by RecyclePivots in the last call
6         ReduceAndExpose(timeslot − RPtime,numtrav,ApplyRule)
7     end
8 end
```

## 4.3 Proof compactness

The focus of this section is the notion of compactness as introduced in §4.1: a proof is compact whenever different nodes are labeled with different clauses, that is $C(n_i) = C(n_j) \implies i = j$ for any $i, j$. We first present an algorithm to address redundancies related to the presence of multiple occurrences of a same unit clause in a proof. Then we illustrate a technique based on a form of structural hashing, which makes a proof more compact by identifying and merging nodes having exactly the same pair of parents. We conclude by showing how to combine these procedures with the Local Transformation Framework.

### *4.3.1 Unit clauses-based simplification*

The simplification of a proof by exploiting the presence of unit clauses has already been addressed in the literature in [30] and [5]. The two works pursue different goals. The *RecycleUnits* algorithm from [5] uses learned unit clauses to rewrite subproofs that were derived before learning them. On the other hand, the *LowerUnits* algorithm from [30] collects unit clauses and reinserts them at the level of the global root, thus removing redundancies due to multiple resolution steps on the same unit clauses.

Following the idea of [30], we present *PushdownUnits*, listed as Algorithm 7. First, the algorithm traverses a proof in a top–down manner, detaching and collecting subproofs rooted in unit clauses, while at the same time reconstructing the proof to keep it legal (based on the schema of Algorithm 2); then, (some of) these subproofs are attached back at the end of the proof, adding new resolution steps. PushdownUnits improves over LowerUnits by performing unit collection and proof reconstruction in a single pass.

The algorithm works as follows. The proof is traversed according to a topological order. When a node $n$ is visited s.t. $C(n)$ is the resolvent of a sound resolution step with pivot $p$, its parents are examined. Assume $n^+$ is a unit clause, that is $C(n^+) = p$; then $n$ is replaced by the other parent $n^-$ and $n^+$ is added to the set of unit clauses $CU$.

This transformation phase might add extra literals $EL$ to the original global root $r$; if this is the case, the necessary resolution steps to make the proof legal are added at the end, starting from the $r$. The nodes previously collected are taken into account one by one; for each $m$, if $C(m) = s$ and $\overline{s}$ is one of the extra literals $EL$, then a new resolution step is added and the new root becomes $m$.

Note that not necessarily all these nodes will be added back to the proof. Multiple nodes might be labeled by the same literal, in which case the correspondent variable will be used only once as pivot. Also, a collected literal which was an antecedent of some resolution step

**Algorithm 7:** PushdownUnits.

**Input**: A legal proof
**Output**: A legal proof
**Data**: $TS$: nodes topological sorting vector; $CU$: collected units set; $EL$: set of extra literals appearing in the global root

```
1  begin
2      TS ← topological_sorting_top_down(proof)
3      r ← global root
4      foreach n ∈ TS do
5          if n is not a leaf then
6              p ← piv(n)
7              if p̄ ∈ C(n⁻) and p ∈ C(n⁺) then
8                  C(n) ← Res_p(C(n⁻), C(n⁺))
9                  if C(n⁺) = p then
10                     Substitute n with n⁻
11                     CU ← CU ∪ {n⁺}
12                 else if C(n⁻) = p̄ then
13                     Substitute n with n⁺
14                     CU ← CU ∪ {n⁻}
15             else if piv(n) ∉ C(n⁻) and piv(n) ∈ C(n⁺) then
16                 Substitute n with n⁻
17             else if piv(n) ∈ C(n⁻) and piv(n) ∉ C(n⁺) then
18                 Substitute n with n⁺
19             else if piv(n) ∉ C(n⁻) and piv(n) ∉ C(n⁺) then
20                 Heuristically choose a parent, substitute n with it;
21     end
22     EL ← extra literals of C(r)
23     foreach m ∈ CU do
24         s ← C(m)
25         if s̄ ∈ EL then
26             Add a new node o s.t. C(o) = Res_{v(s)}(C(r), C(m))
27             r ← o
28     end
29 end
```

might have been anyway resolved upon again along all paths from that resolution step to the global root; if so, it does not appear in the set of extra literals. The subproofs rooted in these unnecessary nodes can be (partially) pruned away to further compress the proof (Fig. 10).

### *4.3.2 Structural hashing*

The work of [21] proposes an algorithm based on a form of *structural hashing*; it explicitly takes into account how resolution proofs are obtained in CDCL SAT-solvers from a sequence of subproofs deriving learnt clauses, and keeps a hash map which stores for each derived clause its pair of antecedents. While building the global proof from the sequence of subproofs, whenever a clause would be added, if its pair of antecedents is already in the hash map, then the existing clause is used.

Taking inspiration from the idea at the base of this technique, we present a post-processing compression algorithm, *StructuralHashing*, which aims at improving the compactness of a proof. StructuralHashing is illustrated in Algorithm 8.

The proof is traversed in topological order. When a node $n$ is visited, the algorithm first checks whether its antecedents are already in the hash map; if so, another node $m$ with the

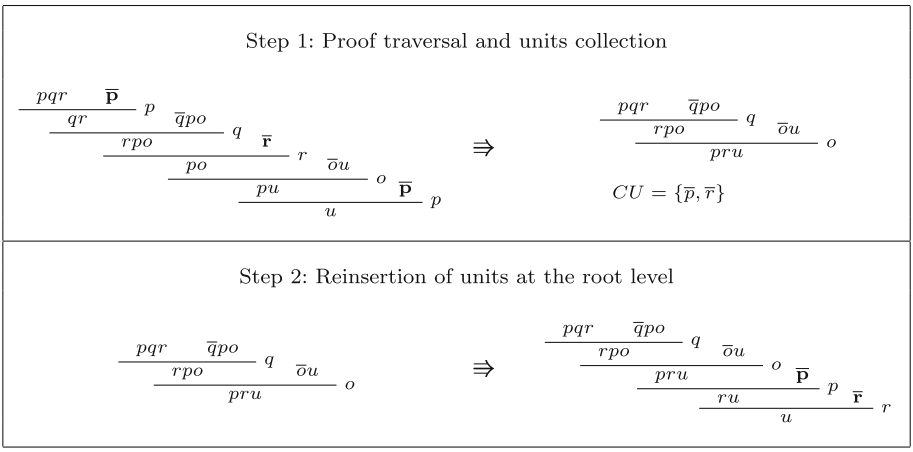



**Fig. 10** Example of application of PushdownUnits. Note that the lowest occurrence of $\overline{p}$ is not added back to the proof

**Algorithm 8:** StructuralHashing.

**Input**: A legal proof
**Output**: A legal proof
**Data**: $TS$: nodes topological sorting vector, $HM$: hash map associating a node to its pair of parents

```
1  begin
2      TS ← topological_sorting_top_down(proof)
3      foreach n ∈ TS do
4          if n is not a leaf then
5              if <n+, n−> ∈ HM then
6                  m ← HM(<n+, n−>)
7                  Replace n with m
8                  Assign n children to m
9              else
10                 HM(<n+, n−>) ← n
11     end
12 end
```

same parents has been seen before. In that case, $n$ is replaced by $m$ and the children of $n$ are assigned to $m$. The use of a topological visit guarantees the soundness of the algorithm: it is safe to replace the subproof rooted in $n$ with that rooted in $m$ since either $(i)$ $m$ is an ancestor of $n$ (and the subproof rooted in $m$ is contained in the subproof rooted in $n$) or $(ii)$ $m$ and $n$ are not on a same path to the global root, so $m$ is not involved in the derivation of $n$.

Note that StructuralHashing does not guarantee a completely compact proof; if two nodes $n_1, n_2$ have the same parents, then $C(n_1) = C(n_2)$, but the converse is not necessarily true. A complete but more computationally expensive technique might consist in employing a hash map to associate clauses with nodes (rather than pairs of nodes with nodes as done in StructuralHashing), based on a function that derives map keys from the clauses content; an implementation of this technique can be found in [63].

#### 4.3.3 StructuralHashing and the Local Transformation Framework

StructuralHashing is a one-pass compression technique, like RecyclePivots and RecyclePivotsWithIntersection. Nevertheless, it is still possible to exploit the Local Transformation Framework in order to disclose new redundancies and remove them, in an iterative manner. We illustrate this approach in Algorithm 9.

**Algorithm 9:** SH + RE.

**Input**: A legal proof; $numloop$: number of global iterations; $numtrav$: number of transformation traversals for each global iteration; $timelimit$: timeout; an instantiation of *ApplyRule*
**Output**: A legal proof

```
1  begin
2      timeslot = timelimit/numloop
3      for i=1 to numloop do
4          StructuralHashing()
5          // SHtime is the time taken by StructuralHashing in the last call
6          ReduceAndExpose(timeslot − SHtime,numtrav,ApplyRule)
7      end
8  end
```

#### 4.3.4 A synergic algorithm

It is possible to combine the compression techniques illustrated so far as shown in Algorithm 10, exploiting their individual features for a synergistic effect. The combined approach executes the algorithms sequentially for a given number of *global iterations*. Note that PushdownUnits is kept outside of the loop: in our experience, SH, RPI and RE are unlikely to introduce unit clauses in the proofs, thus for efficiency PushdownUnits is run only once before the main loop.

**Algorithm 10:** PU + SH + RPI + RE.

**Input**: A legal proof; $numloop$: number of global iterations; $numtrav$: number of transformation traversals for each global iteration; $timelimit$: timeout; an instantiation of *ApplyRule*
**Output**: A legal proof

```
1  begin
2      timeslot = timelimit/numloop
3      PushdownUnits()
4      for i=1 to numloop do
5          StructuralHashing()
6          RecyclePivotsWithIntersection()
7          // SHtime and RPItime are the time taken by StructuralHashing
           // and RecyclePivotsWithIntersection in the last call
8          ReduceAndExpose(timeslot − SHtime − RPItime,numtrav,ApplyRule)
9      end
10 end
```

The overall complexity of the combined algorithm is parametric in the number of global iterations and actual transformation traversals (also depending on the specified time limit).

PushdownUnits performs a topological visit of the proof, collecting unit clauses and adding them back at the level of the global root; the complexity is $O(|V| + |E|)$, linear in the size of the resolution proof DAG.

Complexity is $O(|V| + |E|)$ also for StructuralHashing, which traverses the proof once, making use of an hash table to detect the existence of multiple nodes with the same resolvents.

An iterative implementation of RecyclePivotsWithIntersection consists of a bottom–up scan of the proof, while computing the sets of removable literals and pruning branches, followed by a reconstruction phase; the complexity is again $O(|V| + |E|)$.

Each execution of TransformAndReconstruct, on which ReduceAndExpose is based, computes a topological sorting of the nodes and traverses the proof top–down applying rewriting rules. If $m$ transformation traversals are executed, the complexity of ReduceAndExpose is $O(m(|V| + |E|))$.

Note that PushdownUnits, RecyclePivotsWithIntersection, TransformAndReconstruct also perform operations at the level of clauses, checking the presence of pivots, identifying rule contexts, updating resolvents. These operations depend on the width of the involved clauses; in practice, this value is very small compared to the proof size, and the complexity can be considered $O(|V| + |E|)$.

Finally, if $n$ global iterations are carried out, the total complexity is $O(nm(|V| + |E|))$. There is a clear trade-off between efficiency and compression. The higher the value of $m$ is, the more redundancies are exposed and then removed; in practice, however, especially in case of large proofs, a complexity higher than linear cannot be afforded, so the $nm$ factor should be kept constant in the size of the proofs.

Some heuristics on the application of the local rules in conjunction with RecyclePivots, RecyclePivotsWithIntersection and StructuralHashing have been proved particularly successful: we refer the reader to §4.4, §4.5 and §6 for details.

### 4.4 Experiments on SMT benchmarks

As a first stage of experimentation, we carried out an evaluation of the three algorithms RecyclePivots (RP), ReduceAndExpose (RE), and their combination RP+RE. The algorithms were implemented inside the tool OpenSMT [16], with proof-logging capabilities enabled.

We experimented on the set of unsatisfiable benchmarks taken from the SMT-LIB [54] from the categories QF_UF, QF_IDL, QF_LRA, QF_RDL. For these sets of benchmarks we noticed that the aforementioned compression 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 a 32-bit Ubuntu server featuring a Dual-Core 2 GHz Opteron CPU and 4 GB of memory; a timeout of 600 seconds and a memory threshold of 2GB (whatever is reached first) were put as limit to the executions.

The executions of RE and RP+RE are parameterized with a time threshold, which we set 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 compression. Notice that, regardless of the ratio, RE and RP+RE 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** Results for SMT benchmarks

| (a) | #Bench | | RedNodes% | | RedEdges% | | RedCore% | | TranTime (s) | |
|---|---|---|---|---|---|---|---|---|---|---|
| RP | 1,370 | | 6.7 | | 7.5 | | 1.3 | | 1.7 | |
| (b) | #Bench | | RedNodes% | | RedEdges% | | RedCore% | | TranTime (s) | |
| Ratio | RE | RP+RE | RE | RP+RE | RE | RP+RE | RE | RP+RE | RE | RP+RE |
| 0.01 | 1,364 | 1,366 | 2.7 | 8.9 | 3.8 | 10.7 | 0.2 | 1.4 | 3.5 | 3.4 |
| 0.025 | 1,363 | 1,366 | 3.8 | 9.8 | 5.1 | 11.9 | 0.3 | 1.5 | 3.6 | 3.6 |
| 0.05 | 1,364 | 1,366 | 4.9 | 10.7 | 6.5 | 13.0 | 0.4 | 1.6 | 4.3 | 4.1 |
| 0.075 | 1,363 | 1,366 | 5.7 | 11.4 | 7.6 | 13.8 | 0.5 | 1.7 | 4.8 | 4.5 |
| 0.1 | 1,361 | 1,364 | 6.2 | 11.8 | 8.3 | 14.4 | 0.6 | 1.7 | 5.3 | 5.0 |
| 0.25 | 1,357 | 1,359 | 8.4 | 13.6 | 11.0 | 16.6 | 0.9 | 1.9 | 8.2 | 7.6 |
| 0.5 | 1,346 | 1,348 | 10.4 | 15.0 | 13.3 | 18.4 | 1.1 | 2.0 | 12.1 | 11.5 |
| 0.75 | 1,339 | 1,341 | 11.5 | 16.0 | 14.7 | 19.5 | 1.2 | 2.1 | 15.8 | 15.1 |
| 1 | 1,335 | 1,337 | 12.4 | 16.7 | 15.7 | 20.4 | 1.3 | 2.2 | 19.4 | 18.8 |

#Bench reports the number of benchmarks solved and processed within the time/memory constraints, RedNodes% and RedEdges% report the average compression in the number of nodes and edges of the proof graphs, and RedCore% reports the average compression in the unsatisfiable core size. TranTime is the average transformation time in seconds

**Table 2** Results for SMT benchmarks

| (a) | MaxRedNodes% | | MaxRedEdges% | | MaxRedCore% | |
|---|---|---|---|---|---|---|
| RP | 65.1 | | 68.9 | | 39.1 | |
| (b) | MaxRedNodes% | | MaxRedEdges% | | MaxRedCore% | |
| Ratio | RE | RP+RE | RE | RP+RE | RE | RP+RE |
| 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 |

MaxRedNodes% and MaxRedEdges% are the maximum compression of nodes and edges achieved by the algorithms in the suite on individual benchmarks.

Table 1 shows the average proof compression after the application of the algorithms[1]. Table 1a shows the compression obtained after the execution of RP. Table 1b instead shows the compression obtained with RE and RP+RE parameterized with a timeout (ratio × solving time). In the columns we report the compression in the number of nodes and edges, the

[1] Full experimental data, as well as executables used in tests are available at http://verify.inf.usi.ch/sites/default/files/FMSD2014.tar.gz

compression of the unsatisfiable core, and the actual transformation time. Table 2 is organized as Table 1 except that it reports the best compression values obtained over all the benchmarks suites.

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

The combined approach RP+RE 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 simplifications.

Table 2b displays some remarkable peaks of compression obtained with the RE and RP+RE approaches on the best individual instances. Interestingly we noticed that in some benchmarks, like *24.800.graph* of the QF_IDL suite, RP does not achieve any compression, due to the high amount of nodes with multiple resolvents present in its proof that forces RecyclePivots to keep resetting the removable literals set RL. RP+RE instead, even for a very small ratio (0.01), performs remarkably, yielding 47.6 % compression for nodes, 49.7 % for edges and 45.7 % for core.

### 4.5 Experiments on SAT benchmarks

A second stage of experimentation was preceded by an implementation of all the compression algorithms discussed so far (Algorithms 1–10) within a new tool, PeRIPLO [55]; PeRIPLO, built on MiniSAT 2.2.0, is an open-source SAT-solver which features resolution proof manipulation and interpolant generation capabilities [2].

We evaluated the following algorithms: PushdownUnits (PU), RecyclePivotsWithIntersection (RPI), ReduceAndExpose (RE), StructuralHashing (SH) (Algorithms 3, 4, 7, 8) and their combinations RPI+RE (Algorithm 6), SH+RE (Algorithm 9), PU+RPI+SH+RE (Algorithm 10); the evaluation was carried out on a set of purely propositional benchmarks from the SAT Challenge 2012 [57], the SATLIB benchmark suite [58] and the CMU collection [22].

First, a subset of *unsatisfiable* benchmarks was extracted from the SAT Challenge 2012 collection by running MiniSAT 2.2.0 alone with a timeout of 900 s and a memory threshold of 14 GB; this resulted in 261 instances from the Application track and the Hard Combinatorial track. In addition to these, another 125 unsatisfiable instances were obtained from the SATLIB Benchmark Suite and the CMU collection, for a total of 386 instances.

The experiments were carried out on a 64-bit Ubuntu server featuring a Quad-Core 4GHz Xeon CPU and 16 GB of memory; a timeout of 1,200 s and a memory threshold of 14 GB were put as limit to the executions. The PeRIPLO framework was able to handle proofs up to 30 million nodes, as in the case of the *rbcl_xits_07_UNSAT* instance from the Application track in the SAT Challenge 2012 collection.

Differently from the case of SMT benchmarks, we decided to specify as termination criterion an explicit amount of transformation traversals per global iteration, focusing on the dependency between proofs size and time taken by the algorithms to move over proofs and compress them.

[2] Full experimental data, as well as executables used in tests are available at http://verify.inf.usi.ch/sites/default/files/FMSD2014.tar.gz

**Table 3** Results for SAT benchmarks

| (a) | #Bench | RedNodes% | RedCore% | RedEdges% | TranTime (s) | Ratio |
|---|---|---|---|---|---|---|
| PU | 200 | 1.81 | 0.00 | 2.18 | 5.44 | 0.09 |
| SH | 205 | 5.90 | 0.00 | 6.55 | 4.53 | 0.07 |
| RPI | 203 | 28.48 | 1.75 | 30.66 | 14.32 | 0.21 |
| RE 3 | 203 | 4.16 | 0.09 | 4.85 | 24.84 | 0.31 |
| RE 5 | 203 | 5.06 | 0.14 | 5.88 | 37.86 | 0.41 |
| RE 10 | 202 | 6.11 | 0.17 | 7.08 | 67.09 | 0.56 |
| PU+SH+RPI | 196 | 32.81 | 1.47 | 35.70 | 18.66 | 0.27 |
| (b) RPI+RE | #Bench | RedNodes% | RedCore% | RedEdges% | TranTime (s) | Ratio |
| 2.3 | 201 | 30.69 | 2.08 | 33.49 | 34.78 | 0.39 |
| 2.5 | 200 | 30.71 | 2.15 | 33.53 | 40.37 | 0.45 |
| 3.3 | 200 | 31.28 | 2.23 | 34.22 | 51.43 | 0.51 |
| 3.5 | 200 | 31.56 | 2.34 | 34.50 | 61.16 | 0.56 |
| (c) SH+RE | #Bench | RedNodes% | RedCore% | RedEdges% | TranTime (s) | Ratio |
| 2.3 | 204 | 17.33 | 0.09 | 19.20 | 33.87 | 0.38 |
| 2.5 | 204 | 19.81 | 0.15 | 21.92 | 48.40 | 0.47 |
| 3.3 | 204 | 21.68 | 0.16 | 23.96 | 56.39 | 0.51 |
| 3.5 | 202 | 23.69 | 0.18 | 26.17 | 70.75 | 0.59 |
| (d) PU+SH+RPI+RE | #Bench | RedNodes% | RedCore% | RedEdges% | TranTime (s) | Ratio |
| 2.3 | 195 | 39.46 | 1.89 | 43.34 | 35.23 | 0.44 |
| 2.5 | 195 | 40.46 | 1.93 | 44.49 | 38.49 | 0.46 |
| 3.3 | 195 | 41.68 | 2.06 | 45.86 | 47.41 | 0.51 |
| 3.5 | 195 | 42.41 | 2.05 | 46.71 | 52.91 | 0.54 |

#Bench reports the number of benchmarks solved and processed within the time/memory constraints, RedNodes% and RedEdges% report the average compression in the number of nodes and edges of the proof graphs, RedCore% the average compression in the unsatisfiable core size. TranTime is the average transformation time in seconds; Ratio is the ratio between transformation time and overall time

Table 3 reports the performance of the compression techniques. Table 3a shows the results for the individual techniques PU, SH, RPI, RE, the latter tested for an increasing amount of transformation traversals (3, 5, 10), and the combination PU+SH+RPI without RE. Table 3b–d respectively report on the combinations RPI+RE, SH+RE, PU+SH+RPI+RE: in the first column, a pair $n, m$ indicates that $n$ global iterations and $m$ transformation traversals per global iteration were carried out.

RPI is clearly the most effective technique on a single run, as for compression and ratio transformation time/overall time. For this set of experiments we tuned RE focusing on its ability to disclose new redundancies, so we did not expect exceptional results when running the algorithm by itself; the performance of RE improves with the number of transformation traversals performed, but cannot match that of RPI.

On the other hand, the heuristics adopted in the application of the rewriting rules (see §6) have a major effect on SH, enhancing the amount of compression from about 6 % to more than 20 %.

The combined approaches naturally achieve better and better results as the number of global iterations and transformation traversals grows. In particular, Algorithm 10, which

**Table 4** Results for SAT benchmarks.

| PU+SH+RPI+RE | MaxRedNodes% | MaxRedCore% | MaxRedEdges% |
|---|---|---|---|
| 2.3 | 83.7 | 21.5 | 83.7 |
| 2.5 | 84.9 | 21.6 | 85.2 |
| 3.3 | 87.1 | 22.1 | 87.4 |
| 3.5 | 87.9 | 22.2 | 88.2 |

MaxRedNodes% and MaxRedEdges% are the maximum compression of nodes and edges achieved by the PU+SH+RPI+RE combination on a single benchmark

brings together the techniques for regularization, compactness and redundancies exposure, reaches a remarkable average compression level of 40 %, surpassing (ratio being equal) all other combined approaches.

We report for completeness in Table 4 the maximum compression obtained by the PU+SH+RPI+RE combination on the best individual instances.

A limitation of the current version of PeRIPLO it that preprocessing by SATElite is not enabled in case of proof-logging; this restriction, which entails higher solving times and might yield larger proofs sizes, will be addressed in a future release of the tool.

## 5 Proof transformation for interpolation

Craig interpolants [23], since the seminal work by McMillan [46–48], have been extensively applied in SAT-based model checking and predicate abstraction [39]. Formally, given an unsatisfiable conjunction of formulae $A \wedge B$, an interpolant $I$ is a formula that is implied by $A$ (i.e., $A \implies I$), is unsatisfiable in conjunction with $B$ (i.e., $B \wedge I \implies \bot$) and is defined on the common language of $A$ and $B$. The interpolant $I$ can be thought of as an over-approximation of $A$ that is still in conflict with $B$.

Several state-of-the art approaches exist to generate interpolants in an automated manner; the most successful techniques derive an interpolant for $A \wedge B$ from a proof of unsatisfiability of the conjunction. This approach grants two important benefits: the generation can be achieved in linear time w.r.t. the proof size, and interpolants themselves only contain information relevant to determine the unsatisfiability of $A \wedge B$.

Krajíček and Pudlák [43,53] are probably the first to propose an efficient way to compute interpolants in the context of propositional logic. McMillan [47] proposes an alternative method that also handles the quantifier-free theories of uninterpreted functions, linear arithmetic, and their combination. All these techniques adopt recursive algorithms, which initially set *partial interpolants* for the axioms. Then, following the proof structure, they deduce a partial interpolant for each conclusion from those of the premises. The partial interpolant of the overall conclusion is the interpolant for the formula.

Yorsh and Musuvathi present in [67] a generalization of Pudlák's method that can compute interpolants for a formula defined modulo a theory $\mathcal{T}$. The leaves of the proof of unsatisfiability in this case are original clauses as well as $\mathcal{T}$-lemmata involving original predicates, generated by the prover during the solving process. It is then sufficient to compute a partial interpolant for each theory lemma in order to derive the global interpolant.

The last technique, for its modularity, finds its natural implementation within SMT-solvers [7], procedures that combine SAT-solvers and domain specific algorithms for a theory $\mathcal{T}$ in an efficient way (see §2). Cimatti et al. [20] show that interpolant generation within

SMT-solvers can outperform other known methods (e.g. [47]), as a result of using optimized domain-specific procedures for $\mathcal{T}$.

In the following we use $A$ and $B$ to denote two quantifier-free formulae in a theory $\mathcal{T}$, for which we would like to compute an interpolant. Theories of interest are equality with uninterpreted functions $\mathcal{EUF}$, linear arithmetic over the rationals $\mathcal{LRA}$ and the integers $\mathcal{LIA}$, the theory of arrays $\mathcal{AX}$, or a combination of theories, such as $\mathcal{EUF} \cup \mathcal{LRA}$. Variables that appear only in $A$ or $B$ are called *A-local* and *B-local* respectively. Variables that appear in both $A$ and $B$ are called *AB-common*. A predicate is called *AB-mixed* if it is defined on both $A$-local and $B$-local variables, it is called *AB-pure* otherwise. Notice that $AB$-mixed predicates cannot appear in $A$ and $B$.

*Example 1* Let $A \equiv (x = v \wedge f(x) = z)$, $B \equiv (y = v \wedge f(y) = u \wedge z \neq u)$ be two formulae in the $\mathcal{EUF}$ theory. Variable $x$ is $A$-local, $y$, $u$ are $B$-local, $z$, $v$ are $AB$-common (a predicate $x = y$ would be $AB$-mixed). An interpolant $I$ for $A \wedge B$ is $f(v) = z$, which is an $AB$-pure predicate.

We consider resolution proofs are defined as in §2; recall that propositional variables in a proof may represent the propositional abstraction of theory predicates. In this case we say that a propositional variable is $AB$-mixed if such is the predicate associated with it.

One limitation of the approach of [67] is that theory lemmata, appearing in a proof of unsatisfiability, must not contain $AB$-mixed predicates. However, several decision procedures defined for SMT-solvers heavily rely on the creation of new predicates during the solving process. Examples are delayed theory combination (DTC) [12], Ackermann's expansion [1], lemmas on demand [25] and splitting on demand [6] (see §5.2). All these methods may introduce new predicates, which can potentially be $AB$-mixed.

In this section we show how to compute an $AB$-pure proof from an $AB$-mixed one but without interfering with the internals of the SMT-solver; our technique applies to any approach that requires the addition of $AB$-mixed predicates (see §5.2 for a set of examples). We illustrate how to employ the Local Transformation Framework to effectively modify the proofs, in such a way that the generic method of [67] can be applied; in this way it is possible to achieve a complete decoupling between the solving phase and the interpolant generation phase, provided that an interpolation procedure is available for a conjunction of atoms in $\mathcal{T}$.

A sketch of the approach is depicted in Fig. 11. The idea is to move all $AB$-mixed predicates (in grey) toward the leaves of the proof (Fig. 11b) within maximal $AB$-mixed subproofs.

**Definition 3** (*AB-mixed subproof*) Given a resolution proof $P$, an *AB-mixed subproof* is a subproof $P'$ of $P$ rooted in a clause $C$, whose intermediate pivots are all $AB$-mixed predicates. $P'$ is *maximal* if $C$ does not contain $AB$-mixed predicates.

When dealing with a background theory $\mathcal{T}$ we note the following fact: if $P'$ is a maximal $AB$-mixed subproof rooted in a clause $C$, then *$C$ is a valid theory lemma for $\mathcal{T}$*.

This observation derives from Definition 3 and from the fact that (i) $AB$-mixed predicates can only appear in theory lemmata (as they do not appear in the original formula) and (ii) a resolution step over two theory lemmata generates another theory lemma.

Once $AB$-mixed maximal subproofs are formed, it is possible to replace them with their root clauses (Fig. 11c). The obtained proof is now free of $AB$-mixed predicates and can be used to derive an interpolant applying the method of [67], provided that an interpolant generating procedure is available for the theory $\mathcal{T}$.

The crucial part of our approach is an algorithm for proof transformation. It relies on the Local Transformation Framework discussed in §3. An ad-hoc application of the rules can

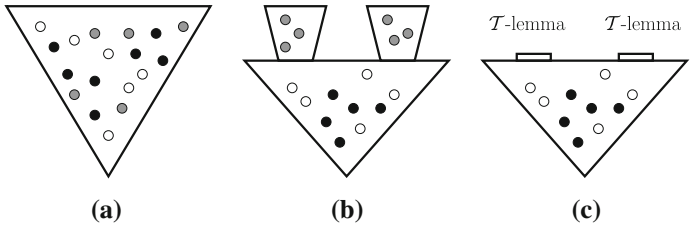



**Fig. 11** An overview of our approach. **a** is the proof generated by the SMT-solver. *White points* represent $A$-local predicates, *black points* represent $B$-local predicates, *grey points* represent $AB$-mixed predicates. b $AB$-mixed predicates are confined inside $AB$-mixed trees. c $AB$-mixed trees are removed and their roots are valid theory lemmata in $\mathcal{T}$

be used to transform a proof $P$ into a proof $P'$, where all $AB$-mixed variables are confined in $AB$-mixed subproofs. Each rewriting rule can effectively swap two pivots $p$ and $q$ in the resolution proof, or perform simplifications, depending on the particular context.

In the following, to facilitate the understanding of the algorithm, we will call $AB$-mixed and $AB$-pure predicates *light* and *heavy* respectively. The rules are applied when a light predicate is below a heavy predicate in the proof graph. The effect of an exhaustive application of the rules is to lift light predicates over heavy predicates as bubbles in water.

### 5.1 Pivot reordering algorithms

The Local Transformation Framework can be effectively employed to perform a *local reordering* of the pivots. Each rule in Fig. 5 either swaps the position of two pivots ($S1$, $S2$, $R2$), or it eliminates at least one pivot ($R1$, $R2'$, $R3$). This feature can be used to create an application strategy aimed at sorting the pivots in a proof $P$, by transforming it into a proof $P'$ such that all light variables are moved above heavy variables.

In order to achieve this goal it is sufficient to consider only *unordered* contexts, i.e. those in which $v(t)$ is a light variable and $v(s)$ is a heavy variable. Therefore a simple non-deterministic algorithm can be derived as Algorithm 11.

**Algorithm 11:** PivotReordering.

**Input**: A legal proof
**Output**: A legal proof without unordered contexts
**Data**: $U$: set of unordered contexts
1 **begin**
2  Determine $U$, e.g. through a visit of the proof
3  **while** $U \neq \emptyset$ **do**
4  Choose a context in $U$
5  Apply the associated rule, and SubsumptionPropagation if necessary
6  Update $U$
7  **end**
8 **end**

The algorithm terminates: note in fact that each iteration strictly decreases the distance of an occurrence of a heavy pivot w.r.t. the global root, until no more unordered contexts are left.

**Algorithm 12:** ApplyRuleForPivotReordering.

```
   Input: A left context lc, a right context rc
1  begin
2      if lc is ordered and rc is unordered then
3          Apply rule for rc
4      else if lc is unordered and rc is ordered then
5          Apply rule for lc
6      else if lc is unordered and rc is unordered then
7          Heuristically choose between lc and rc and apply rule
8  end
```

A more efficient choice is to make use of Algorithm 2 TransformAndReconstruct, by instantiating the ApplyRule method so that it systematically pushes light variables above heavy ones; a possible instantiation is shown in Algorithm 12. An algorithm for pivot reordering would then consist of a number of consecutive runs of TransformAndReconstruct, stopping when no more unordered contexts are found: Algorithm 13, *PivotReordering2*, implements this approach.

**Algorithm 13:** PivotReordering2.

```
   Input: A legal proof
   Output: A legal proof without unordered contexts
1  begin
2      while unordered contexts are found do
3          TransformAndReconstruct(ApplyRuleForPivotReordering)
4      end
5  end
```

## 5.2 SMT-solving and $AB$-mixed predicates

In this section we show a number of techniques currently employed in state-of-the-art SMT-solvers that can potentially introduce $AB$-mixed predicates during the solving phase. If these predicates become part of the proof of unsatisfiability, the proof reordering algorithms described in §5.1 can be applied to produce an $AB$-pure proof.

### *5.2.1 Theory reduction techniques*

Let $\mathcal{T}_k$ and $\mathcal{T}_j$ be two decidable theories such that $\mathcal{T}_k$ is weaker (less expressive) than $\mathcal{T}_j$. Given a $\mathcal{T}_j$-formula $\varphi$, and a decision procedure SMT($\mathcal{T}_k$) for quantifier-free formulae in $\mathcal{T}_k$, it is often possible to obtain a decision procedure SMT($\mathcal{T}_j$) for quantifier-free formulae in $\mathcal{T}_j$ by augmenting $\varphi$ with a finite set of $\mathcal{T}_k$-lemma $\psi$. These lemmata (or axioms) explicitly encode the necessary knowledge such that $\mathcal{T}_k \models \varphi \wedge \psi$ if and only if $\mathcal{T}_j \models \varphi$. Therefore a simple decision procedure for $\mathcal{T}_j$ is as described by Algorithm 14.

In practice the lemmata generation function can be made lazy by plugging it inside the SMT-solver directly; this paradigm is known as lemma on demand [25] or splitting on demand [6]. We show some reduction techniques as follows.

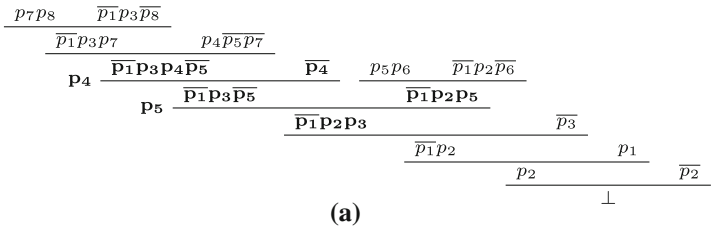
*Reduction of $\mathcal{AX}$ to $\mathcal{EUF}$* We consider the case where $\mathcal{T}_k \equiv \mathcal{EUF}$, the theory of equality with uninterpreted functions, and $\mathcal{T}_j \equiv \mathcal{AX}$, the theory of arrays with extensionality. The

**Algorithm 14:** A reduction approach for SMT($\mathcal{T}_j$).

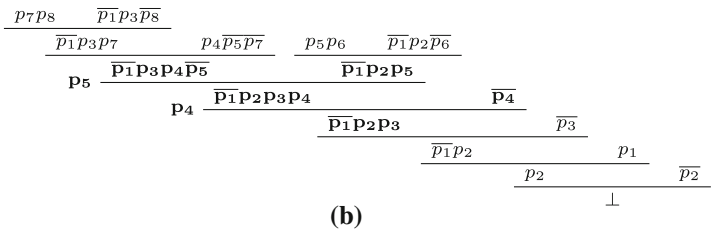
**Input**: $\varphi$ for $\mathcal{T}_j$

```
1 begin
2     ψ = generateLemmata(φ)
3     return SMT(T_k)(φ ∧ ψ)
4 end
```

| Id | Clauses | Prop. abstract. |
|---|---|---|
| 1 | $x = wr(y, i, e)$ | $p_1$ |
| 2 | $rd(x, j) \neq rd(y, j)$ | $\overline{p_2}$ |
| 3 | $rd(x, k) \neq rd(y, k)$ | $\overline{p_3}$ |
| 4 | $j \neq k$ | $\overline{p_4}$ |
| 5 | $(i = j \vee rd(wr(y, i, e), j) = rd(y, j))$ | $p_5\, p_6$ |
| 6 | $(i = k \vee rd(wr(y, i, e), k) = rd(y, k))$ | $p_7\, p_8$ |
| 7 | $(x \neq wr(y, i, e) \vee rd(x, j) = rd(y, j) \vee rd(wr(y, i, e), j) \neq rd(y, j))$ | $\overline{p_1}\, p_2\, \overline{p_6}$ |
| 8 | $(x \neq wr(y, i, e) \vee rd(x, k) = rd(y, k) \vee rd(wr(y, i, e), k) \neq rd(y, k))$ | $\overline{p_1}\, p_3\, \overline{p_8}$ |
| 9 | $(j = k \vee i \neq j \vee i \neq k)$ | $p_4\, \overline{p_5}\, \overline{p_7}$ |




**(a)**




**(b)**

**Fig. 12** Clauses from Example 2. $\varphi \equiv \{1, 2, 3, 4\}$, $\psi \equiv \{5, 6\}$. Clauses 7–9 are theory lemmata discovered by the $\mathcal{EUF}$ solver. **a** Is a possible proof obtained by the SMT-solver (for $\mathcal{EUF}$) on $\varphi \wedge \psi$. **b** Is a proof after swapping $p_4$ and $p_5$ by means of rule *S*2; in the resulting proof all mixed literals ($p_5$–$p_8$) appear in the upper part of the proof in an $AB$-mixed proof subtree. The root of the $AB$-mixed subtree $\overline{p_1}\, p_2\, p_3\, p_4$ is a valid theory lemma in $\mathcal{AX}$

axioms of $\mathcal{EUF}$ are the ones of equality (reflexivity, symmetry, and transitivity) plus the congruence axioms $\forall x, y.\ x = y \implies f(x) = f(y)$, for any functional symbol of the language.

The theory of arrays $\mathcal{AX}$ is instead axiomatized by:

$$\forall x, i, e.\ \ rd(wr(x, i, e), i) = e \tag{1}$$

$$\forall x, i, j, e.\ \ i = j \vee rd(wr(x, i, e), j) = rd(x, j) \tag{2}$$

$$\forall x, y.\ \ x = y \iff (\forall i.\ rd(x, i) = rd(y, i)) \tag{3}$$

State-of-the-art approaches for $\mathcal{AX}$ implemented in SMT-solvers [10,14,24,32] are all based on reduction to $\mathcal{EUF}$. Instances of the axioms of $\mathcal{AX}$ are added to the formula in a lazy manner until either the formula is proven unsatisfiable or saturation is reached. The addition of new lemmata may require the creation of $AB$-mixed predicates when a partitioned formula is considered.

*Example 2* Let $\varphi \equiv A \wedge B$, where $A \equiv x = wr(y,i,e)$, and $B \equiv rd(x,j) \neq rd(y,j) \wedge rd(x,k) \neq rd(y,k) \wedge j \neq k$. Variables $\{i,e\}$ are $A$-local, $\{j,k\}$ are $B$-local, and $\{x,y\}$ are $AB$-common. To prove $\varphi$ unsatisfiable with a reduction to $\mathcal{EUF}$, we need to instantiate axiom (2) twice as $\psi \equiv (i = j \vee rd(wr(y,i,e),j) = rd(y,j)) \wedge (i = k \vee rd(wr(y,i,e),k) = rd(y,k))$. Notice that we introduced four $AB$-mixed predicates. Now we can send $\varphi \wedge \psi$ to an SMT-solver for $\mathcal{EUF}$ to produce the proof of unsatisfiability. Figure 12 shows a possible resolution proof generated by the SMT-solver, and how it can be transformed into a proof without $AB$-mixed predicates.

*Reduction of $\mathcal{LIA}$to $\mathcal{LRA}$* Decision procedures for $\mathcal{LIA}$ (linear integer arithmetic) often rely on iterated calls to a decision procedure for $\mathcal{LRA}$ (linear rational arithmetic). An example is the method of *branch-and-bound*: given a feasible rational region $R$ for a set of variables $\mathbf{x} = (x_1, \ldots, x_n)$, and a non-integer point $\mathbf{c} \in R$ for $\mathbf{x}$, then one step of branch-and-bound generates the two subproblems $R \cup \{x_i \leq \lfloor c_i \rfloor\}$ and $R \cup \{x_i \geq \lceil c_i \rceil\}$. These are again recursively explored until an integer point $\mathbf{c}$ is found.

Note that the splitting on the bounds can be delegated to the propositional engine by adding the lemma $((x_i \leq \lfloor c_i \rfloor) \vee (x_i \geq \lceil c_i \rceil))$. In order to obtain a faster convergence of the algorithm, it is possible to split on *cuts*, i.e. linear constraints, rather than on simple bounds. However cuts may add $AB$-mixed predicates if $A$-local and $B$-local variables are mixed into the same cut.

*Example 3* Let $\varphi \equiv A \wedge B$ in $\mathcal{LIA}$, where $A \equiv 5x - y \leq 1 \wedge y - 5x \leq -1$, and $B \equiv 5z - y \leq -2 \wedge y - 5z \leq 3$. The axiom $\psi \equiv ((x - z \leq 0) \vee (x - z \geq 1))$ (which contains two $AB$-mixed literals) is sufficient for $\varphi \wedge \psi$ to be proven unsatisfiable by a solver for $\mathcal{LRA}$, by discovering two additional theory lemmata $((5x - y \not\leq 1) \vee (y - 5z \not\leq 3) \vee (x - z \leq 0))$ and $((5x - y \not\leq -1) \vee (y - 5z \not\leq -2) \vee (x - z \geq 1))$.

*Ackermann's Expansion* When $\mathcal{T}_j$ is a combination of theories of the form $\mathcal{EUF} \cup \mathcal{T}_k$, Ackermann's expansion [1] can be used to reduce the reasoning from $\mathcal{T}_j$ to $\mathcal{T}_k$. The idea is to use as $\psi$ the exhaustive instantiation of the congruence axiom $\forall x, y\ (x = y \implies f(x) = f(y))$ for all pairs of variables appearing in uninterpreted functional symbols and all uninterpreted functional symbols $f$ in $\varphi$. This instantiation generates $AB$-mixed predicates when $x$ is instantiated with an $A$-local symbol and $y$ with a $B$-local one.

*Example 4* Let $\mathcal{T}_k \equiv \mathcal{LRA}$. Let $\varphi \equiv A \wedge B$ and $A \equiv (a = x + y \wedge f(a) = c)$, $B \equiv (b = x + y \wedge f(b) = d \wedge c \neq d)$. The axiom $\psi \equiv ((a \neq b) \vee (f(a) = f(b))$ is sufficient for $\mathcal{LRA}$ to detect the unsatisfiability of $\varphi \wedge \psi$, by discovering two additional theory lemmata $((f(a) \neq f(b)) \vee (f(a) \neq c) \vee (f(b) \neq d) \vee (c \neq d))$ and $((a \neq x + y) \vee (b \neq x + y) \vee (a = b))$.

### *5.2.2 Theory combination via DTC*

A generic framework for theory combination was introduced by Nelson and Oppen in [51]. We recall it briefly as follows.

Given two signature-disjoint and stably-infinite theories $\mathcal{T}_1$ and $\mathcal{T}_2$, a decision procedure for a conjunction of constraints in the combined theory $\mathcal{T}_1 \cup \mathcal{T}_2$ can be obtained from the decision procedures for $\mathcal{T}_1$ and $\mathcal{T}_2$. First, the formula $\varphi$ is *flattened*, i.e. auxiliary variables are introduced to separate terms that contain both symbols of $\mathcal{T}_1$ and $\mathcal{T}_2$. Then the idea is that the two theory solvers for $\mathcal{T}_1$ and $\mathcal{T}_2$ are forced to exhaustively exchange *interface equalities* i.e. equalities between *interface variables* (interface variables are those that appear both in constraints of $\mathcal{T}_1$ and $\mathcal{T}_2$ after flattening).[3]

DTC implements a non-deterministic version of the Nelson–Oppen framework, in which interface equalities are not exchanged by the deciders directly, but they are guessed by the SAT-solver. With DTC it is possible to achieve a higher level of modularity w.r.t. the classical Nelson–Oppen framework. DTC is currently implemented (with some variations) in most state-of-the-art SMT-solvers.

If no $AB$-mixed interface equality is generated, an interpolant can be derived with the methods already present in the literature; otherwise our method can be applied to reorder the proof, as an alternative to the techniques described in [20,33].

*Example 5* Consider again $\varphi$ of Ex. 4. Since $a$, $b$, $f(a)$, $f(b)$ appear in constraints of both theories, we need to generate two interface equalities $a = b$ and $f(a) = f(b)$. The guessing of their polarity is delegated to the SAT-solver. The SMT-solver will detect the unsatisfiability after the $\mathcal{EUF}$-solver discovers the two theory lemmata $((a \neq b) \vee (f(a) = f(b))$ and $((f(a) \neq f(b)) \vee (f(a) \neq c) \vee (f(b) \neq d) \vee (c \neq d))$ and the $\mathcal{LRA}$-solver discovers the theory lemma $((a \neq x + y) \vee (b \neq x + y) \vee (a = b))$.

### 5.3 Experiments on SMT benchmarks

For the purpose of this experimentation we chose to focus on one particular application among those of §5.2, namely Ackermann's expansion for theory combination.

We evaluated the proof transformation technique on the set of `QF_UFIDL` formulae from the SMT-LIB [54] (`QF_UFIDL` refers to the combined theory $\mathcal{EUF} \cup \mathcal{IDL}$). The suite contains 319 unsatisfiable instances. Each formula was split in half to obtain an artificial interpolation problem (in the same fashion as [20].)[4]

The pivot reordering algorithm Algorithm 13 was realized by means of the Local Transformation Framework and implemented in OpenSMT [16]. Proof manipulation was applied when the proof contained $AB$-mixed predicates, in order to lift them up inside $AB$-maximal subproofs and replace them with their roots.

We ran the experiments on a 32-bit Ubuntu server equipped with Dual-Core 2 GHz Opteron 2212 CPU and 4 GB of memory. The benchmarks were executed with a timeout of 60 min and a memory threshold of 2 GB (whatever was reached first): 172 instances, of which 82 proofs contained $AB$-mixed predicates,[5] were successfully handled within these limits. We have reported the cost of the transformation and its effect on the proofs; the results are summarized in Table 5. We grouped benchmarks together following the original classification used in SMT-LIB and provided average values for each group [4].

The results in Table 5 demonstrate that our proof transformation technique induces, on average, about 13 % overhead with respect to plain solving time. The average increase in

[3] Note that in practice flattening can be avoided. For instance in Example 5 we do not perform any flattening.

[4] The benchmarks and the detailed results are available at http://verify.inf.usi.ch/sites/default/files/FMSD2014.tar.gz

[5] Notice that in some cases $AB$-mixed predicates were produced during the search, but they did not appear in the proof.

**Table 5** The effect of proof transformation on QF_UFIDL benchmarks summarized per group: #Bench—number of benchmarks in a group, #$AB$—average number of $AB$-mixed predicates in a proof, Time%—average time overhead induced by transformation, Nodes% and Edges%—average difference in the proof size as a result of transformation

| Group | #Bench | #$AB$ | Time% | Nodes% | Edges% |
|---|---|---|---|---|---|
| RDS | 2 | 7 | 84 | −16 | −19 |
| EufLaArithmetic | 2 | 74 | 18 | 187 | 193 |
| pete | 15 | 20 | 16 | 66 | 68 |
| pete2 | 52 | 13 | 6 | 73 | 80 |
| uclid | 11 | 12 | 29 | 87 | 90 |
| Overall | 82 | 16 | 13 | 74 | 79 |

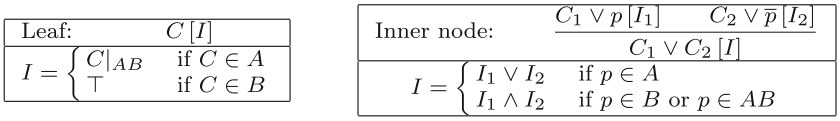



**Fig. 13** McMillan interpolation algorithm

size is around 74 %, but not all the instances experienced a growth; we observed in fact that in 42 out of 82 benchmarks the transformed proof was smaller than the original one both in the number of nodes and edges. Overall it is important to point out that the creation of new nodes due to the application of the $S$ rules did not entail any exponential blow-up in the size of the proofs during the transformation process.

Another interesting result to report is the fact that only 45% of the proofs contained $AB$-mixed predicates and, consequently, required transformation. This is another motivation for using off-the-shelf algorithms for SMT-solvers and have the proof transformed in a second stage, rather than tweaking (and potentially slowing down) the solver to generate clean proofs upfront.

### 5.4 Pivot reordering for propositional interpolation

This section concludes our discussion on interpolation by moving back from the context of SMT to that of SAT. We complete the analysis begun by the authors of [42] and illustrate how, in the case of purely propositional refutations, a transformation technique can be devised to generate interpolants directly in conjunctive or disjunctive normal form.

Assuming a refutation of a formula $A \wedge B$, we distinguish whether a variable $p$ is local to $A$ ($p \in A$), local to $B$ ($p \in B$) or common to $A$ and $B$ ($p \in AB$). Figure 13 shows McMillan interpolation algorithm for propositional logic [28,47]. The algorithm initially sets a partial interpolant for the clauses that label the refutation leaves; in particular, the partial interpolant of a clause in $A$ is its restriction $C|_{AB}$ to the propositional variables common to $A$ and $B$. Then, recursively, a partial interpolant for each resolvent is computed from those of the antecedents depending on whether the pivot appears only in $A$ ($I_1 \vee I_2$) or not ($I_1 \wedge I_2$); the partial interpolant of the global root is the interpolant for $A \wedge B$. In Fig. 13, $C[I]$ means that clause $C$ has a partial interpolant $I$. $I_1$, $I_2$ and $I$ are the partial interpolants respectively associated with the two antecedents $C_1 \vee p$, $C_2 \vee \overline{p}$ and the resolvent $C_1 \vee C_2$ of a resolution step.

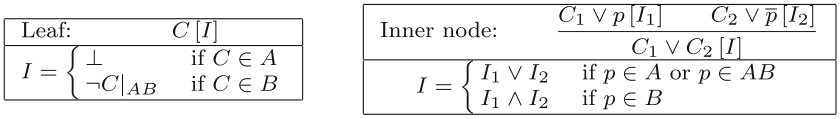



**Fig. 14** McMillan′ interpolation algorithm

Algorithm 13, PivotReordering2, can be employed to restructure a refutation so that McMillan interpolation algorithm generates an interpolant in CNF. It is sufficient in fact to modify the definition of light and heavy predicates given in §5, so that a context is considered unordered whenever $v(t)$ is local to $A$ (*light*) and $v(s)$ is a propositional variable in $B$ or in $AB$ (*heavy*). Effect of the proof transformation is to push up light variables, so that, along every path from the leaves to the root, light variables appear before heavy variables.

We need to show that this condition is sufficient in order for McMillan algorithm to produce an interpolant in CNF.

**Theorem 3** *Assume a refutation $P$ without unordered contexts. McMillan interpolation algorithm generates an interpolant in CNF from $P$.*

*Proof (by structural induction)*

***Base case*** The partial interpolant for a leaf labeled by a clause $C$ is either $\top$ or $C|_{AB}$, so it is in CNF.

***Inductive step*** Given an inner node $n$ and the associated pivot $p = piv(n)$, assume the partial interpolants $I_1$ and $I_2$ for $C(n^+) = C_1 \vee p$ and $C(n^-) = C_2 \vee \overline{p}$ are in CNF. We have four possibilities:

- Case 1: $I_1$ and $I_2$ are both in clausal form; then either $n^+$, $n^-$ are leaves or they are inner nodes with light pivot variables. $p$ can be either light or heavy: in the first case $I$ is itself a clause, in the second case $I$ is a conjunction of clauses, so it is in CNF.
- Case 2: $I_1$ is a clause, $I_2$ is a conjunction of at least two clauses; then $n^+$ can be either a leaf or an inner node with a light pivot, but $I_2$ must be an inner node with a heavy pivot (due to $\wedge$ being the main connective of $I_2$). Since $P$ does not have unordered contexts, $p$ must be a heavy variable, thus $I = I_1 \wedge I_2$ is in CNF.
- Case 3: $I_1$ is a conjunction of at least two clauses, $I_2$ is a clause. Symmetric to Case 2.
- Case 4: Both $I_1$ and $I_2$ are a conjunction of at least two clauses. As for Case 2 and Case 3.

A similar argumentation holds for the generation of interpolants in disjunctive normal form. Let us consider the algorithm dual to McMillan, which we address as McMillan′ [28], illustrated in Fig. 14.

Algorithm 13 can be employed to transform the refutation; in this case a context is unordered if $v(t)$ is a variable local to $B$ (*light*) and $v(s)$ is a variable local to $A$ or shared (*heavy*). The effect of pushing up light variables is that, during the construction of the interpolant, the connective $\wedge$ will be introduced before $\vee$ along each path, so that the resulting interpolant will be in disjunctive normal form (note that the partial interpolant of a leaf is already in DNF, being a conjunction of literals).

We can thus state the following theorem:

**Theorem 4** *Assume a refutation $P$ without unordered contexts. McMillan′ interpolation algorithm generates an interpolant in DNF from $P$.*

As already pointed out in [42], the price to pay for a complete transformation might be an exponential increase of the proof size, due to the node duplications necessary to apply rules

$S1$, $S2$, $R2$ to contexts where $C4$ has multiple children (see Fig. 5). A feasible compromise consists in performing a partial CNFization or DNFization by limiting the application of such rules to when $C4$ has a single child; in this case, the proof growth depends only on the application of rule $S1$, and the increase is maintained linear.

## 6 Heuristics for the proof transformation algorithms

In this section we discuss some of the heuristics implemented in OpenSMT and PeRIPLO to guide the application of the Local Transformation Framework rules and the reconstruction of proofs, with reference to compression (§4) and pivot reordering for interpolation (§5).

Some of the algorithms presented so far (Algorithms 1, 2, 7) need to handle the presence of resolution steps which are not valid anymore since the pivot is missing from both antecedents; in that case, the resolvent node $n$ must be replaced by either parent. A heuristics which has been proven useful for determining the replacing parent is the following. If one of the parents (let us say $n^+$) has only $n$ as child, then $n$ is replaced by $n^-$; since $n^+$ loses its only child, then (part of) the subproof rooted in $n^+$ gets detached from the global proof, yielding a simplification of the proof itself. If both parents have more than one child, then the parent labeled by the smaller clause is the one that replaces $n$, aiming at increasing the amount of simplifications performed while moving down to the global root.

As far as the heuristics for the application of rewriting rules are concerned, the ApplyRule method adheres to some general lines. Whenever a choice is possible between a left and a right context, a precedence order is respected: ($X > Y$ means: the application of $X$ is preferred over that of $Y$):

$$R3 > \{R2', R1\} > R2 > S1' > S2 > S1$$

The compression rules $R$ have always priority over the shuffling rules $S$, $R3$ being the favorite, followed by $R2'$ and $R1$. Among the $S$ rules, $S1'$ is able to perform a local simplification, which makes it preferred to $S2$ and especially to $S1$, which increases the size of the proof; between equal $S$ rules, the one which does not involve a node duplication (see Fig. 5) is chosen.

Additional constraints depend on the actual goal of the transformation. If the aim is pivot reordering, the constraints are as illustrated in Algorithm. 13, with ties broken according to the general lines given above. If the aim is compression, then $S1$ is never applied, since it increases the size of the proof and it is not apparent at the time of its application whether it would bring benefits in a second moment, neither are applied $R2$, $S1'$, $S2$ if they involve a duplication. A strategy which proved successful in the application of $S$ rules is to push up nodes with multiple resolvents whenever possible, with the aim of improving the effect of RecyclePivots and RecyclePivotsWithIntersection; interestingly, this technique shows as a side effect the disclosure of redundancies which can effectively be taken care of by StructuralHashing.

These heuristics have been discovered through experimentation and have been adopted due to their practical usefulness for compression, in a setting where the large size of proofs allows only a few traversals (and thus a limited application of rules) by means of ReduceAndExpose, and where the creation of new nodes should be avoided; it is thus unlikely that, arbitrarily increasing the number of traversals, they would expose and remove all pivots redundancies. A more thorough, although practically infeasible, approach could rely on keeping track of all contexts and associated rules in a proof $P$. Since the $S$ rules are revertible, an equivalence relation $\equiv_S$ could be defined among proofs so that $P \equiv_S P'$ if $P'$ can be obtained from $P$ (and vice versa) by means of a sequence of applications of $S$ rules. A backtracking-based

algorithm could be employed to systematically visit equivalence classes of proofs, and to move from an equivalence class to another thanks to the application of an $R$ rule.

## 7 Related work

Various proof manipulation techniques have been developed in the last years, the main goal being compression.

In [2], Amjad proposes an algorithm based on heuristically reordering the resolution steps that form a proof, trying to identify a subset of the resolution steps that is still sufficient to derive the empty clause. The approach relies on using an additional graph-like data structure to keep track of how literals of opposite polarity are propagated from the leaves through the proof and then resolved upon.

Sinz [61] explicitly assumes a CDCL context, where a resolution-based SAT-solver generates a sequence of derivations called *proof chains*, combined in a second moment to create the overall proof. He presents an algorithm that works at the level of proof chains, aiming at identifying and merging shared substructures to generate a smaller proof.

Amjad further develops this approach in [3]. He adopts a representation of resolution proofs that allows the use of efficient algorithms and data structures for substring matching; this feature is exploited to perform memoization of proofs by detecting and reusing common subproofs.

Cotton introduces in [21] two compression methods. The first one is based on a form of structural hashing, where each inner node in a proof graph is associated with its pair of antecedents in a hash map. The compression algorithm traverses the sequence of proof chains while updating the hash map, and adds a resolution step to the overall proof only if it does not already exist. The second one consists of a rewriting procedure that, given in input a proof and a heuristically chosen propositional variable $p$, transforms the proof so that the last resolution step is on $p$; this might result in a smaller proof.

Bar-Ilan et al. [5] present a technique that exploits learned unit clauses to rewrite subproofs that were derived before learning them. They also propose a compression algorithm (*RecyclePivots*) that searches for resolution steps on the same pivot along paths from leaves to the root in a proof. If a pivot is resolved upon more than once on a path (which implies that the pivot variable is introduced and then removed multiple times), the resolution step closest to the root is kept, while the others are simplified away. The algorithm is effective on resolution proof trees, but can be applied only in a limited form to resolution proof DAGs, due to the possible presence of multiple paths from a node to the root.

This restriction is relaxed in the work of Fontaine et al. [30], who extend the algorithm of [5] into *RecyclePivotsWithIntersection* to keep track, for each node, of the literals which get resolved upon along *all* paths from the node to the root. [30] also presents an algorithm that traverses a proof, collecting unit clauses and reinserting them at the level of the global root, thus removing redundancies due to multiple resolution steps on the same unit clauses; this technique is later generalized in [11] to lowering subproofs rooted in non-unit clauses.

[38] builds upon [5] in developing three variants of *RecyclePivots* tailored to resolution proof DAGs. The first one is based on the observation that the set of literals which get resolved in a proof upon along all paths from the node to the root must be a superset of the clause associated to the node, if the root corresponds to the empty clause. The second and third ones actually correspond respectively to *RecyclePivotsWithIntersection* and to a parametric version of it where the computation of the set of literals is limited to nodes with up to a certain amount of children.

Our set of compression techniques has been illustrated with reference to [5,61] and [30] in §4.

Besides compression, a second area of application of proof manipulation has been interpolation, both in the propositional and in the first order settings.

D'Silva et al. [27] introduce a global transformation framework for interpolation to reorder the resolution steps in a proof with respect to a given partial order among pivots; compression is shown to be a side effect for some benchmarks. Compared to [27], our approach works locally, and leaves more freedom in choosing the strategies for rule applications. Also our target is not directly computing interpolants, but rather rewriting the proof in such a way that existing techniques can be applied.

The same authors focus in [28] on the concept of strength of an interpolant. They present an analysis of existing propositional interpolation algorithms, together with a method to combine them in order to obtain weaker or stronger interpolants from a same proof of unsatisfiability. They also address the use and the limitations of the local transformation rules of Jhala and McMillan [42]. The rewriting rules corresponding to $S1$ and $S2$ in the Local Transformation Framework (§3) were first introduced in [42] and further examined in [28] as a way to modify a proof to obtain stronger or weaker interpolants, once fixed the interpolation algorithm; we devised the remaining rules after an exhaustive analysis of the possible proof contexts. [42] also discusses the application of $S1$ and $S2$ to generate interpolants in conjunctive normal form; however, not all the contexts are taken into account, and, as pointed out in [28], the contexts for $S1$ and $S2$ are not correctly identified.

Note that $S1$ and $S2$ have also a counterpart in Gentzen's sequent calculus system $LK$ [31]: $S1$ corresponds to swapping applications of the structural cut and contraction rules, while $S2$ is one of the rank reduction rules.

Interpolation for first order theories in presence of $AB$-mixed predicates is addressed in [20], only for the case of DTC, by tweaking the decision heuristics of the solver, in such a way that it guarantees that the produced proof can be handled with known methods. In particular the authors define a notion of *ie-local proofs*, and they show how to compute interpolants for this class of proofs, and how to adapt an SMT-solver to produce only $ie$-local proofs. [33] the relaxes the constraint on generating $ie$-local proofs by introducing the notion of *almost-colorable* proofs. We argue that our technique is simpler and more flexible, as different strategies can be derived with different applications of our local transformation rules. Our method is also more general, since it applies not only to theory combination but to any approach that requires the addition of $AB$-mixed predicates (see §5.2).

More recently, a tailored interpolation algorithm has been proposed in [18] for the combined theory of linear arithmetic and uninterpreted functions; it has the notable feature of allowing the presence of mixed predicates, thus making proof manipulation not necessary anymore.

*Clausal Proofs.* This paper addresses resolution proofs in the context of transformation for compression and Craig interpolation; state-of-the-art algorithms, as described in the previous sections, rely on representing and manipulating proofs in the form of directed acyclic graphs. However, alternative approaches exist; for example, CDCL SAT-solvers can be instrumented to generate proofs in *clausal format*, as a sequence of learned clauses [34,40,65]. The development of compression techniques tailored to clausal proofs is an interesting topic, which will be investigated as future work.

## 8 Conclusions

In this paper we have presented a proof transformation framework based on a set of local rewriting rules and shown how it can be applied to the tasks of proof compression and pivot reordering.

As for compression, we discussed how rules that effectively simplify the proof can be interleaved with rules that locally perturbate the topology, in order to create new opportunities for compression. We identified two kinds of redundancies in proofs, related to the notions of regularity and compactness, and presented and compared a number of algorithms to address them, moving from existing techniques in the literature. Individual algorithms, as well as their combinations, were implemented and tested over a collection of benchmarks both from SAT and SMT libraries, showing remarkable levels of compression in the proof size.

As for pivot reordering, we described how to employ the rewriting rules to isolate and remove $AB$-mixed predicates, in such a way that standard procedures for interpolation in SMT can be applied. The approach enables the use of off-the-shelf techniques for SMT-solvers that are likely to introduce $AB$-mixed predicates, such as Ackermann's expansion, lemma on demand, splitting on demand and DTC. We showed by means of experiments that our rules can effectively transform the proofs without generating any exponential growth in their size. Finally, we explored a form of interaction between LISs and proof manipulation by providing algorithms to reorder resolution steps in a propositional proof to guarantee the generation of interpolants in conjunctive or disjunctive normal form.

## References


1. Ackermann W (1954) Solvable cases of the decision problem. Studies in logic and the foundations of mathematics. North-Holland, Amsterdam
2. Amjad H (2007) Compressing propositional refutations. Electron Notes Theor Comput Sci 185:3–15
3. Amjad H (2008) Data compression for proof replay. J Autom Reason 41(3–4):193–218
4. Amla N, McMillan K (2003) Automatic abstraction without counterexamples. In: TACAS, pp 2–17
5. Bar-Ilan O, Fuhrmann O, Hoory S, Shacham O, Strichman O (2008) Linear-time reductions of resolution proofs. In: HVC, pp 114–128
6. Barrett C, Nieuwenhuis R, Oliveras A, Tinelli C (2006) Splitting on demand in SAT modulo theories. In: LPAR, pp 512–526
7. Barrett C, Sebastiani R, Seshia S, Tinelli C (2009) Satisfiability modulo theories. In: Biere A, Heule M, van Maaren H, Walsh T (eds) Handbook of satisfiability. IOS Press, Amsterdam, pp 825–885
8. Bayardo RJ, Schrag R (1997) Using CSP look-back techniques to solve real-world SAT instances. In: AAAI/IAAI, pp 203–208
9. Biere A, Cimatti A, Clarke E, Strichman O, Zhu Y (2003) Bounded model checking. Adv Comput 58:117–148
10. Bofill M, Nieuwenhuis R, Oliveras A, Rodrguez-Carbonell E, Rubio A (2008) A write-based solver for SAT modulo the theory of arrays. In: FMCAD, pp 101–108
11. Boudou J, Paleo B (2013) Compression of propositional resolution proofs by lowering subproofs. In: TABLEAUX, pp 237–251
12. Bozzano M, Bruttomesso R, Cimatti A, Junttila T, Ranise S, van Rossum P, Sebastiani R (2005) Efficient satisfiability modulo theories via delayed theory combination. In: CAV, pp 335–349
13. Bradley AR (2011) SAT-based model checking without unrolling. In: VMCAI, pp 70–87
14. Brummayer R, Biere A (2008) Lemmas on demand for the extensional theory of arrays. In: Workshop on SMT
15. Bruni R (2003) Approximating minimal unsatisfiable subformulae by means of adaptive core search. Discret Appl Math 130(2):85–100
16. Bruttomesso R, Pek E, Sharygina N, Tsitovich A (2010) The OpenSMT Solver. In: TACAS, pp 150–153
17. Bruttomesso R, Rollini S, Sharygina N, Tsitovich A (2010) Flexible interpolation with local proof transformations. In: ICCAD, pp 770–777

18. Christ J, Hoenicke J, Nutz A (2013) Proof tree preserving interpolation. In: TACAS, pp 124–138
19. Cimatti A, Griggio A, Sebastiani R (2007) A simple and flexible way of computing small unsatisfiable cores in SAT modulo theories. In: SAT, pp 334–339
20. Cimatti A, Griggio A, Sebastiani R (2008) Efficient interpolant generation in satisfiability modulo theories. In: TACAS, pp 397–412
21. Cotton S (2010) Two techniques for minimizing resolution proofs. In: SAT, pp 306–312
22. CMU Benchmarks. http://www.cs.cmu.edu/~modelcheck/bmc/bmc-benchmarks.html. Accessed 24 April 2014
23. Craig W (1957) Three uses of the herbrand–gentzen theorem in relating model theory and proof theory. J Symb Log 22(3):269–285
24. de Moura L, Bjørner N (2009) Generalized, efficient array decision procedures. In: FMCAD, pp 45–52
25. de Moura L, Rue H (2002) Lemmas on demand for satisfiability solvers. In: SAT, pp 244–251
26. Dershowitz N, Hanna Z, Nadel A (2006) A scalable algorithm for minimal unsatisfiable core extraction. In: SAT, pp 36–41
27. D'Silva V, Kroening D, Purandare M, Weissenbacher G (2008) Restructuring resolution refutations for interpolation. Technical report, ETH
28. D'Silva V, Kroening D, Purandare M, Weissenbacher G (2010) Interpolant strength. In: VMCAI, pp 129–145
29. Fontaine P, Marion J, Merz S, Nieto L, Tiu A (2006) Expressiveness + automation + soundness: towards combining SMT solvers and interactive proof assistants. In: TACAS, pp 167–181
30. Fontaine P, Merz S, Paleo B (2011) Compression of propositional resolution proofs via partial regularization. In: CADE, pp 237–251
31. Gentzen G (1935) Untersuchungen über das logische schließen. i. Math Z 39(1):176–210
32. Goel A, Krstić S, Fuchs A (2008) Deciding array formulas with frugal axiom Instantiation. In: SMT, pp 12–17
33. Goel A, Krstić S, Tinelli C (2009) Ground interpolation for combined theories. In: CADE, pp 183–198
34. Goldberg E, Novikov Y (2003) Verification of proofs of unsatisfiability for CNF formulas. In: DATE, pp 10,886–10,891
35. Gomes C, Kautz H, Sabharwal A, Selman B (2008) Satisfiability solvers. In: van Harmelen F, Lifschitz V, Porter B (eds) Handbook of knowledge representation. Elsevier, Amsterdam, pp 89–134
36. Grégoire E, Mazure B, Piette C (2007) Local-search extraction of muses. Constraints 12(3):325–344
37. Grumberg O, Lerda F, Ofer OS, Theobald M (2005) Proof-guided underapproximation-widening for multi-process systems. In: POPL, pp 122–131
38. Gupta A (2012) Improved single pass algorithms for resolution proof reduction. In: ATVA, pp 107–121
39. Henzinger T, Jhala R, Majumdar R, McMillan K (2004) Abstractions from proofs. In: POPL, pp 232–244
40. Heule M, Hunt W, Wetzler N (2013) Trimming while checking clausal proofs. In: FMCAD
41. Huang J (2005) Mup: a minimal unsatisfiability prover. In: ASP-DAC, pp 432–437
42. Jhala R, McMillan K (2005) Interpolant-based transition relation approximation. In: CAV, pp 39–51
43. Krajíček J (1997) Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. J Symb Log 62(2):457–486
44. Lynce I, Marques-Silva J (2004) On computing minimum unsatisfiable cores. In: SAT, pp 305–310
45. Marques-Silva J, Sakallah K (1996) GRASP—a new search algorithm for satisfiability. In: ICCAD, pp 220–227
46. McMillan K (2003) Interpolation and SAT-based model checking. In: CAV, pp 1–13
47. McMillan K (2004) An interpolating theorem prover. In: TACAS, pp 16–30
48. McMillan K (2004) Applications of Craig interpolation to model checking. In: CSL, pp 22–23
49. Mneimneh M, Lynce I, Andraus Z, Marques-Silva J, Sakallah K (2005) A branch-and-bound algorithm for extracting smallest minimal unsatisfiable formulas. In: SAT, pp 467–474
50. Necula G (1997) Proof-carrying code. In: POPL, pp 106–119
51. Nelson G, Oppen D (1979) Simplification by cooperating decision procedures. ACM Trans Progr Lang Syst 1(2):245–257
52. Oh Y, Mneimneh MN, Andraus ZS, Sakallah KA, Markov IL (2004) AMUSE: a minimally-unsatisfiable subformula extractor. In: DAC, pp 518–523
53. Pudlák P (1997) Lower bounds for resolution and cutting plane proofs and monotone computations. J Symb Log 62(3):981–998
54. Ranise S, Tinelli C The satisfiability modulo theories library (SMT-LIB). http://www.smtlib.org. Accessed 24 April 2014
55. Rollini S Proof transformer and interpolator for propositional logic (PeRIPLO). http://verify.inf.usi.ch/content/periplo. Accessed 24 April 2014

56. Rollini S, Bruttomesso R, Sharygina N (2010) An efficient and flexible approach to resolution proof reduction. In: HVC, pp 182–196
57. SAT Challenge (2012) http://baldur.iti.kit.edu/SAT-Challenge-2012/. Accessed 24 April 2014
58. SATLIB Benchmark Suite http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html . Accessed 24 April 2014
59. Sebastiani R (2007) Lazy satisfiability modulo theories. JSAT 3:144–224
60. Shlyakhter I, Seater R, Jackson D, Sridharan M, Taghdir M (2003) Debugging overconstrained declarative models using unsatisfiable cores. In: ASE, pp 94–105
61. Sinz C (2007) Compressing propositional proofs by common subproof extraction. In: EUROCAST, pp 547–555
62. Sinz C, Kaiser A, Kuchlin W (2003) Formal methods for the validation of automotive product configuration data. AI EDAM 17(1):75–97
63. Skeptik Proof Theory Library https://github.com/Paradoxika/Skeptik. Accessed 24 April 2014
64. Tseitin GS (1968) On the complexity of derivation in the propositional calculus. In: Slisenko AO (ed) Studies in constructive mathematics and mathematical logic. Plenum, New York, pp 115–125
65. Van Gelder A (2008) Verifying RUP proofs of propositional unsatisfiability. In: ISAIM
66. Weber T, Amjad H (2009) Efficiently checking propositional refutations in hol theorem provers. J Appl Log 7(1):26–40
67. Yorsh G, Musuvathi M (2005) A combination method for generating interpolants. In: CADE, pp 353–368
68. Zhang L, Malik S (2003) Extracting small unsatisfiable cores from unsatisfiable Boolean formulas. In: SAT
69. Zhang L, Sharad M (2003) Validating SAT solvers using an independent resolution-based checker: practical implementations and other applications. In: DATE, pp 10,880–10,885