On Extending Bounded Proofs to Inductive Proofs

Oded Fuhrmann and Shlomo Hoory

IBM Haifa Research Lab
{odedf,shlomoh}@il.ibm.com

Abstract. We propose a method for extending a bounded resolution proof to an unbounded inductive proof. More specifically, given a resolution proof that a state machine beginning at an initial state satisfies some property at cycle k, we show that the existence of a Δ-invariant cut implies that the property holds for cycles $k+\Delta$, $k+2\Delta$, etc. We suggest a linear algorithm for identifying such Δ-extendible proofs and develop the required theory for covering all cycles by Δ-extendible proofs. To expose Δ-invariant cuts, we develop an efficient proof manipulation algorithm that rearranges the proof by the natural temporal order. We demonstrate the applicability of our techniques on a few real-life examples.

Keywords: Formal verification, resolution proofs, extending proofs, proof simplification.

1 Introduction

One of the main focal points of formal verification is proving bounded and unbounded safety properties. Namely, given a non-deterministic finite state machine, one would like to know whether all states reachable from an initial state set I comply with a specifying property S. The bounded problem asks if the above holds for a specific cycle k, while the unbounded problem asks if the property holds for all non-negative values of k. An efficient algorithm to either problem seems hopeless, since the problems are NP-complete and PSPACE-complete, respectively [SC85].

The safety problem has attracted a significant amount of research due to its practical importance. The common state-of-the-art technology for solving this problem is based on an invocation of a DPLL-based SAT solver. Various SAT-based approaches have been suggested over the last decade, such as SAT-based BMC [BCCZ99], k-induction [SSS00], interpolation [McM03], and proof-based abstraction refinement [LWS03]. These solutions utilize the fact that, given a SAT query, any modern solver returns either a satisfying assignment or a resolution-based refutation that no such assignment exists (see [ZM03b]).

In this work, we propose a method for extending the proof that a property holds at cycle k to an infinite number of cycles[1]. The method consists of an

[1] An interesting reference in this context is [SK97] in which a method for sequential equivalence based on unwinding of the circuit combined with learning is presented.

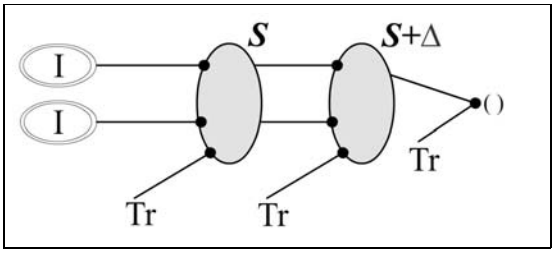


Fig. 1. Δ-extendible proof: The gray Δ-invariant cut separates the root from the encircled Init axioms and is sufficient to prove its Δ shift, possibly using Transition Relation Axioms

efficient algorithm that can be added on top of a standard BMC platform, as an extra post-processing stage for the generated proofs. When successful, our algorithm declares that the proof for the property at cycle k is Δ-extendible, implying the property at cycles $k + i\Delta$, for all positive integers i.

A proof is Δ-extendible if it contains a Δ-invariant cut. That is, a set of proof nodes S that separates the Init axioms from the consequence of the proof, and that implies its forward shift by Δ cycles. This is demonstrated in Figure 1. One should note that $\Delta = 1$ implies that the property holds for all cycles starting at k. However, the problem of covering all cycles when $\Delta > 1$ requires careful treatment.

We mention, for the reader acquainted with Interpolation Theory [McM03] that the Δ-invariant cut defined above is an interpolant. This is implied by the fact that it partitions the proof into two disjoint sets of clauses (A, B), where A includes the cut. This special interpolant, in conjunction with transition relation constraints, implies the same interpolant shifted Δ cycles. The interested reader is referred to [FH09] for an example of a Δ-extendible proof where the classical interpolant computation [McM03] leads to a spurious solution.

As might be expected, looking for a Δ-invariant cut in a messy computer generated proof is rather naive. We suggest an algorithm for preprocessing the proof to reveal Δ-invariant cuts. The algorithm consists of two components. The novel component reorders the proof by increasing cycle number from its axioms to the final consequent clause. The second component is the double pivot simplification, which was introduced in [BIFH$^+$08]. Such an algorithm may be of interest in its own right for any other application using resolution proofs, such as building better interpolants [JM05] and minimizing resolution proofs [GKS08, ZM03a]. We emphasize that both preprocessing components, as well as the detection of Δ-extendible cuts are efficient in that they amount to a linear pass of the proof, though their success is not guaranteed.

The experimental results of this work include a few examples taken from the IBM internal benchmark suite. Starting with the smallest benchmark problems, our technique solved the first three problems, but failed on the fourth. For each example, our algorithm was given a sequence of SAT solver generated proofs, where the k'th proof implied the property at cycle k. For the first three examples, our algorithm deduced the property for all k, where two extensions had $\Delta = 2$, which amounted to covering the even and odd values of k by two separate proofs. In all successful examples preprocessing was paramount. Reporting the negative result for the fourth example is important since it exposes the weak sides of our method and points to possible directions for future research.

The rest of the paper is organized as follows: In Section 2, we give an example for our method and introduce the necessary notation. In Section 3, we show how to extend proofs by Δ-invariant cuts, present an algorithm for finding such a cut, and discuss the problem of covering all cycles when $\Delta > 1$. In Section 4, we present the proof reorder algorithm. Section 5 deals with the experimental results, and Section 6 concludes.

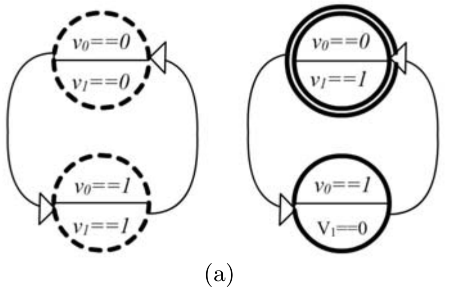
2 A Motivating Example, and Some Notation

2.1 Preliminaries and Notations

A literal is a Boolean variable or its negation, and a clause is a set of literals. We say that a clause is satisfied if at least one of its literals is assigned true. Given two clauses c_1, c_2 and a variable p s.t. $p \in c_1$ and $-p \in c_2$ their resolution is defined as $res_p(c_1, c_2) := (c_1 \cup c_2) \setminus \{p, -p\}$ and the variable p is called the resolution pivot. A proof $\mathcal{P}(A, R)$ is a directed acyclic graph on the nodes $A \cup R$ corresponding to axioms and resolution nodes. An *axiom node* $n \in A$ holds a clause field $n.C$. A *resolution node* $n \in R$ holds the clause field $n.C$, two references to other nodes $n.L, n.R$, and the pivot literal $n.Piv$. The edges of the dag are $(n.L, n)$ and $(n.R, n)$ for all $n \in R$. The fields $n.C$ and $n.Piv$ are defined so that $n.C = res_{n.Piv}(n.L.C, n.R.C)$ for all $n \in R$. We use the notation $S_1 \underset{\mathcal{P}}{\vdash} S_2$ when the proof $\mathcal{P}(A, R)$ with $A \subseteq S_1$ proves a set of clauses $S_2 \subseteq \{n.C \text{ for } n \in A \cup R\}$. If S_2 consists of a single clause, we call it the consequence of $\mathcal{P}$ and the node of the clause the root, $root(\mathcal{P})$.[2]

We assume that the finite state machine has an n bit state $(v_1, \ldots, v_n)$, where v_i is a binary state signal and v_i^j represents the state signal v_i at cycle j. The set $\{v_i^j\}_{i,j}$ is the problem variables. The cycle shift operator $(\cdot)_\Delta$ shifts its argument by Δ cycles forward, and is defined on variables, literals, clauses, etc. For example, $(\{-v_1^5, v_2^3\})_3 = \{-v_1^8, v_2^6\}$. A model M for a state machine is a pair of sets of clauses (M_I, M_{Tr}) describing the Init and Transition Relation constraints encoded into clauses (see [BCCZ99, Tse68]). The set M_I describes the initial constraints, while M_{Tr} describes the transition relation of the underlying state machine. We assume that M_I is finite and that M_{Tr} is the closure under the cycle shift operator of a finite clause set $base(M_{Tr})$. The negation of the property at cycle k is encoded as the clauses $M_{\overline{Spec_k}}$. We will usually assume that $M_{\overline{Spec_k}}$ consists of the single literal $-s^k$. By the notation above, $M \underset{\mathcal{P}}{\vdash} s^k$ states that $\mathcal{P}$ proves that the property holds at cycle k.

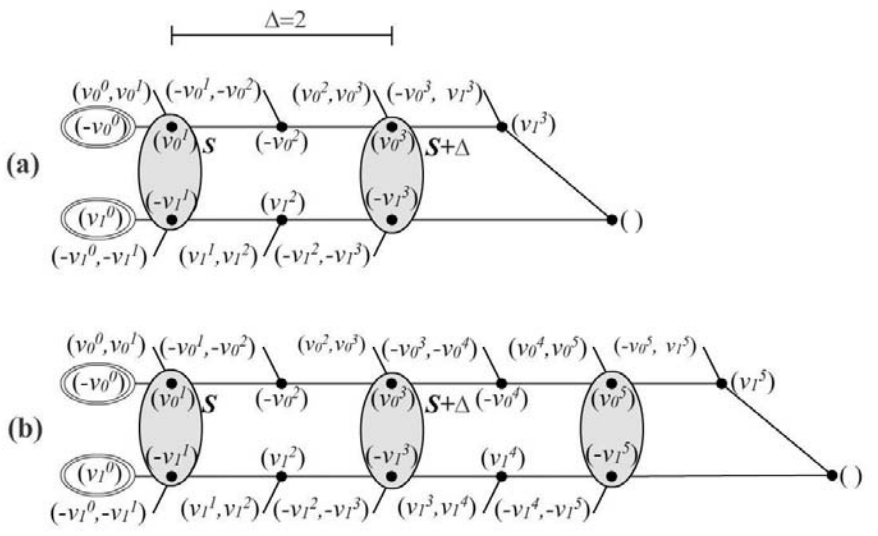
[2] In the sequel, we abuse notation and identify a set of proof nodes with their clauses.



$$
\begin{aligned}
M_I &= \{\, (-v_0^0), (v_1^0)\} \\
M_{Tr} &= \{\, (v_0^0, v_0^1), (-v_0^0, -v_0^1), \\
&\qquad (v_1^0, v_1^1), (-v_1^0, -v_1^1), \\
&\qquad \vdots \\
&\qquad (v_0^{k-1}, v_0^k), (-v_0^{k-1}, -v_0^k), \\
&\qquad (v_1^{k-1}, v_1^k), (-v_1^{k-1}, -v_1^k)\} \\
M_{\overline{Spec_k}} &= \{\, (v_0^k, -v_1^k), (-v_0^k, v_1^k)\}
\end{aligned}
$$

(b)

Fig. 2. A state machine: (a) represented as a state diagram, (b) encoded as CNF



Δ=2
(a)
(v0^0,v0^1) (-v0^1,-v0^2) (v0^2,v0^3) (-v0^3, v1^3)
(-v0^0) (v0^1) S (-v0^2) (v0^3) S+Δ (v1^3)
(v1^0) (-v1^1) (v1^2) (-v1^3) ( )
(-v1^0,-v1^1) (v1^1,v1^2) (-v1^2,-v1^3)
(b)
(v0^0,v0^1) (-v0^1,-v0^2) (v0^2,v0^3) (-v0^3,-v0^4) (v0^4,v0^5) (-v0^5, v1^5)
(-v0^0) (v0^1) S (-v0^2) (v0^3) S+Δ (-v0^4) (v0^5) (v1^5)
(v1^0) (-v1^1) (v1^2) (-v1^3) (v1^4) (-v1^5) ( )
(-v1^0,-v1^1) (v1^1,v1^2) (-v1^2,-v1^3) (v1^3,v1^4) (-v1^4,-v1^5)

Fig. 3. (a) A proof that *Spec* holds at cycle 3 for the state machine of Figure 2a, where S is a 2-invariant cut, (b) an extension of the proof, using the invariant S, showing that *Spec* holds at cycle 5

2.2 An Example

Consider the state machine described in Figure 2a. Its state is described by two binary state variables v_1 and v_2, yielding a state space of size 4. The single initial state, (double encircled) is $(v_0 = 0, v_1 = 1)$, while the states violating the specified property $\overline{Spec} = \{(v_0 = 0, v_1 = 0), (v_0 = 1, v_1 = 1)\}$ are dashed. It is visually apparent that no “unsafe” finite path exists.

In the bounded variant of the above problem one would like to know if there is a finite path of length k from an init state to $\overline{Spec}$. The model of the problem, which is given in Figure 2b, is passed to a SAT solver that checks if there exists a satisfying assignment for all given clauses. As expected, since the problem has no satisfying assignment, the solver supplies us with a refutation showing that the state machine encoded in Figure 2b conforms with the given *Spec* at cycle k. An example for such proof when $k = 3$ is given in Figure 3a. The essence of this paper is that such a proof can be extended to all odd $k \geq 3$. This is demonstrated by a proof for $k = 5$ (see Figure 3b). This extension is possible since there is a grayed 2-invariant cut. In the following section we address this topic rigorously.

3 Proof Extension

Given a proof for the spec at cycle k, one would like to extend it to other cycles in a similar manner to the familiar inductive proof process. The following definition and theorem give a sufficient condition for achieving this.

Definition 1 (Δ-invariant cut). *Let $\mathcal{P}$ be a resolution proof for s^k from $M_I \cup M_{Tr}$. Then, a set of proof nodes S is a Δ-invariant cut if (a) all paths in $\mathcal{P}$ from an I axiom to root($\mathcal{P}$) go through S, and (b) $S \cup M_{Tr} \vdash (S)_\Delta$. A Δ-extendible proof is a proof with a Δ-invariant cut.*

Theorem 1 (Proof Extension Theorem). *If there is a Δ-extendible proof for s^k, then there is a Δ-extendible proof for $s^{k+\Delta}$.*

Corollary 1 (Inductive extension). *If there is a Δ-extendible proof for the spec at cycle k, then the spec holds for all cycles $k, k+\Delta, k+2\Delta, \ldots$.*

Proof (of Theorem 1). We first mention the composition of implication: If $M_1 \underset{\mathcal{P}_a}{\vdash} M_2$ and $M_2 \cup M_1 \underset{\mathcal{P}_b}{\vdash} M_3$ then $M_1 \underset{\mathcal{P}_b \circ \mathcal{P}_a}{\vdash} M_3$, where the proof $\mathcal{P}_b \circ \mathcal{P}_a$ is obtained by taking disjoint copies of $\mathcal{P}_a$ and $\mathcal{P}_b$, and identifying the M_2 axiom nodes of $\mathcal{P}_b$ with the corresponding nodes in $\mathcal{P}_a$. One should note that M_2 is a cut between M_1 and M_3 in the resulting proof.

Given some proof $\mathcal{P}$ for s^k with a Δ-invariant cut S:

(1)	$M_I \cup M_{Tr} \underset{\mathcal{P}_1}{\vdash} S$	since S is a set of proof nodes
(2)	$M_{Tr} \cup S \underset{\mathcal{P}_2}{\vdash} (S)_\Delta$	since S is Δ-invariant
(3)	$M_{Tr} \cup S \underset{\mathcal{P}_3}{\vdash} s^k$	since S is a cut
(4)	$M_I \cup M_{Tr} \underset{\mathcal{P}_2 \circ \mathcal{P}_1}{\vdash} (S)_\Delta$	by composition of (1),(2)
(5)	$M_{Tr} \cup (S)_\Delta \underset{(\mathcal{P}_3)_\Delta}{\vdash} s^{k+\Delta}$	(3) shifted forward by Δ cycles
(6)	$M_I \cup M_{Tr} \underset{(\mathcal{P}_3)_\Delta \circ \mathcal{P}_2 \circ \mathcal{P}_1}{\vdash} s^{k+\Delta}$	by composition of (2),(5).

Therefore, $\mathcal{P}' = (\mathcal{P}_3)_\Delta \circ \mathcal{P}_2 \circ \mathcal{P}_1$ is a proof for the spec at cycle $k + \Delta$. It remains to show that S is a Δ-invariant cut for $\mathcal{P}'$. Indeed, S is a cut between $\mathcal{P}_1$ and $(\mathcal{P}_3)_\Delta \circ \mathcal{P}_2$ as observed earlier. Also, $S \cup M_{Tr} \vdash (S)_\Delta$ since S is a Δ-invariant cut for $\mathcal{P}$. This completes the proof of the theorem. $\square$

Let us recall that the main objective of this work is to prove that the spec holds at all cycles. In view of Theorem 1, the obvious approach is as follows: As long as the spec is not known to hold at all cycles, ask the SAT solver to prove the spec for the smallest unknown cycle, analyze the proof hoping to find a Δ-invariant cut for some Δ, and declare that the spec holds for the appropriate set of cycles. The following two algorithms `ProofExtension` and `AnalayzeProof` make the above framework precise.

Algorithm 1. `ProofExtension`

```
Input: Models $M_I, M_{Tr}$, and the spec signal $s$
Output: Fail at $k$ or Passed

1.  $U \leftarrow \{0, 1, 2, \ldots\}$
2.  While $U \neq \emptyset$
3.     $k \leftarrow \min(U)$
4.     $(res, \mathcal{P}) \leftarrow$ SatSolve($k$)
5.     If $res ==$ SAT
6.        Return Fail at $k$
7.     Else
8.        $\Delta \leftarrow$ AnalayzeProof($\mathcal{P}, k$)
9.        $U \leftarrow U \setminus \{k, k+\Delta, k+2\Delta, \ldots\}$
10. Return Passed
```

Algorithm 2. `AnalyzeProof`

```
Input: A proof $\mathcal{P}$ proving the spec at cycle $k$
Output: The smallest $\Delta$ for which $\mathcal{P}$ is $\Delta$-extendible

1.  For $\Delta$ in {1,...,k}
2.     If DeltaExtendible($\mathcal{P}, \Delta$)
3.        Return $\Delta$
4.  Return 0
```

One should note that there are two problems with the complete realization of the plan. The first is the infinite set of uncovered cycles U used by Algorithm `ProofExtension`. The second and more substantial problem is that of checking if a proof is Δ-extendible. We discuss the two problems in the following subsections.

3.1 Dealing with an Infinite Set

Consider the infinite set U of uncovered cycles used by Algorithm 1. After m iterations, the covered cycles are described by m pairs of numbers (k_i, Δ_i) for $i = 1, \ldots, m$, where (k_i, Δ_i) is the value of (k, Δ) at the end of the i'th iteration. Each such pair (k, Δ) represents an arithmetic progression $k, k+\Delta, k+2\Delta, \ldots$, where for $\varDelta = 0$ the progression reduces to a single number k. Although the required query for the minimal uncovered cycle in U seems to demand searching an infinite set, we show that it suffices to search among a finite number of candidate. These candidates are $k_m + 1, k_m + 2, \ldots, k_m + L$, where L is the least common multiple (lcm) of the non-zero integers in $\varDelta_1, \ldots, \varDelta_m$. The algorithm and correctness lemma are stated next. One should note that the efficiency of the algorithm depends on L, which can conceivably be large. However the largest value we have encountered in practice is $L = 2$, as shown in the experimental section.

Algorithm 3. `FindFirstUncovered`

```
Input: A sequence $(k_i, \varDelta_i)$ for $i = 1, \ldots, m$ supplied by Algorithm 1
Output: The smallest uncovered cycle, or $\infty$ if no such cycle

1. Let $L$ be the lcm of the non-zero integers in $\varDelta_1, \ldots, \varDelta_m$
2. For $c$ in $\{k_m + 1, k_m + 2, \ldots, k_m + L\}$
3.    Mark $c$ as uncovered
4.    For $i$ in $\{1, \ldots, m\}$
5.       If $\varDelta_i > 0$ and $(c \equiv k_i \mod \varDelta_i)$
6.          Mark $c$ as covered
7. Return the minimal uncovered $c$ or $\infty$ if no such $c$
```

Lemma 1. *Algorithm* `FindFirstUncovered` *is correct when supplied with* $(k_i, \varDelta_i)$ *pairs from Algorithm 1.*

3.2 Detecting $\varDelta$-Extendible Proofs

We suggest a naive algorithm `DeltaExtendible` that identifies a subset of the $\varDelta$-Extendible proofs. The algorithm performs DFS on the proof DAG starting from the root, backtracking when it reaches an axiom, a previously visited node, or a clause whose forward shift by $\varDelta$ cycles was already visited. The algorithm returns true, if no init axiom was reached. The correctness of the algorithm follows from the fact that when it returns true, one can construct a $\varDelta$-invariant cut for the proof. The cut is just the set of all nodes n where the DFS backtracked because $(n.C).\varDelta$ was already visited. The correctness of the algorithm is stated in Lemma 2 below.

Algorithm `DeltaExtendible` can be implemented efficiently, since it consists of a linear pass over the proof, where the maintenance and queries to the set of previously visited clauses V is performed in logarithmic time to the number of proof nodes. Further optimization can be achieved by ignoring nodes that are not tagged by init (i.e. that no init axiom was used in their proof). Another question that should be raised is about the best order of visiting proof nodes, which can be any topological order from root to axioms. It is not difficult to construct synthetic $\varDelta$-extendible proofs that are recognized by one order, but not by another. This direction of work requires more research.

Algorithm 4. `DeltaExtendible`

```
Input: A proof $\mathcal{P}$ proving the spec at cycle $k$, and a shift $\Delta$
Output: If true, then $\mathcal{P}$ is $\Delta$-Extendible

1.  Mark all proof nodes as unvisited
2.  Let $V \leftarrow \emptyset$
3.  Return RecDeltaExtendible($root(\mathcal{P}), \Delta, V$)
```

Algorithm 5. `RecDeltaExtendible`

```
Input: A proof node $n$, a shift $\Delta$, and a reference to a set of clauses $V$

1.  If ($n$ is marked visited) or ($n \in M_{Tr}$) or $(n.C)_\Delta \in V$
2.      Return True
3.  Else if $n \in M_I$
4.      Return False
5.  Else
6.      Mark $n$ visited
7.      Let $V \leftarrow V \cup \{n.C\}$
8.      Return RecDeltaExtendible($n.L, \Delta, V$)
            and RecDeltaExtendible($n.R, \Delta, V$)
```

Lemma 2. *If* `DeltaExtendible`$(root(\mathcal{P}), \Delta)$ *returns true then* $\mathcal{P}$ *is* Δ*-extendible.*

4 Proof Reordering

The proof extension algorithm described in the previous section implicitly assumes that a significant portion of the proofs generated by the SAT solver satisfy the necessary conditions of Theorem 1. As might be expected, our experiments indicate that this assumption is overly optimistic in practice. The obvious way to proceed is *preprocessing.* This section explores a few preprocessing algorithms that can be applied to real-life proofs. The techniques explored in this section use the *double pivot* simplification method, (see [BIFH$^+$08]). Double pivot simplification identifies cases where the same pivot variable was used more than once on a path from the root to an axiom and attempts to eliminate them.

4.1 On Combs and Order

Proofs generated by industrial SAT solvers tend to have special properties that may be exploited. In particular, such proofs consist of proof segments representing the derivation of conflict clauses, which are paths with internal nodes of out-degree one. We call such a proof segment *a comb.* Since combs are relatively isolated proof segments, and since many combs have a significant size, it is natural to try and optimize combs.

Formally, a comb $\overline{C}$ in the proof P is a directed path $(n_1, n_2, \ldots, n_k)$ in the DAG of P such that outdegree$(n_i) = 1$ for all $i < k$. For ease of exposition, we always consider left handed combs where $n_{i+1}.L = n_i$ for all i. The comb resolves the clauses $c_0 = n_1.L.C, c_1 = n_1.R.C, c_2 = n_2.R.C, \ldots, c_k = n_k.R.C$ to obtain $n_k.C$. Since the actual nodes used by the comb are immaterial to the exposition, we denote $\overline{C} = (c_0, c_1, \ldots, c_k)$, and $res(\overline{C}) = n_k.C$. The support of a comb, $supp(\overline{C}) = \{c_0, c_1, \ldots, c_k\}$, is its unordered set of clauses.

We attempt to optimize a comb by rearranging its clauses so that pivots are ordered by their cycle number whenever possible. The following example demonstrates the power of this technique. It can be easily verified that the resulting comb satisfies the condition of Theorem 1. Consider the model:

$$\begin{aligned} M_I &= \{\{x^0\}, \{y^0\}\} \\ base(M_{Tr}) &= \{\{-x^0, -y^0, x^1\}, \{-y^0, y^1\}, \{-x^0, -y^0, s^0\}\} \\ M_{\overline{Spec_k}} &= \{-s^k\}. \end{aligned}$$

The comb in Figure 4a proves that the spec holds at cycle 4. Rearranging the comb results in significantly shorter clauses as can be seen in Figure 4b. In fact, extending this example to a comb proving s^n yields clauses of width $\Theta(n)$ before rearranging, and $\Theta(1)$ after. Moreover, the rearranged comb satisfies the requirements of Theorem 1 while the original comb did not. It is interesting to compare the rather synthetic example with real-life results, as depicted in Figures 4, 6 respectively. In the next subsection, we describe the theoretical and algorithmic framework needed for reordering combs.

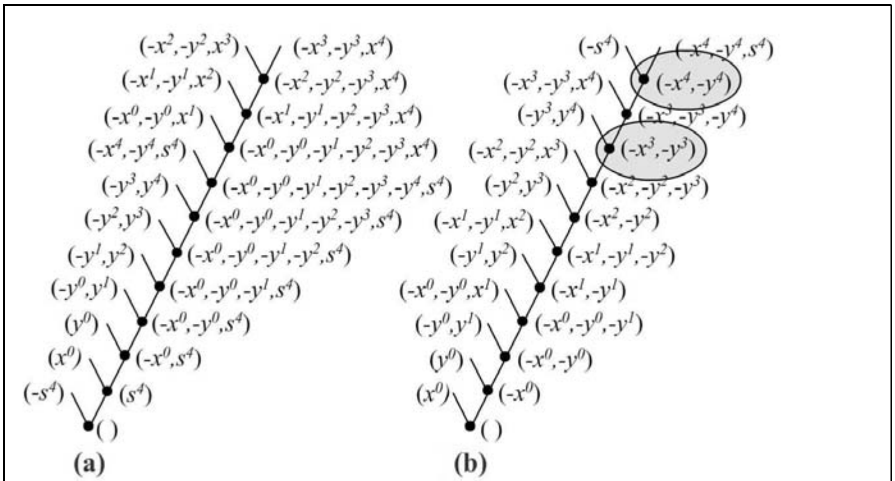


Fig. 4. A comb before and after proof reordering according to literals cycles

4.2 The CombReorder Algorithm

This subsection is devoted to the development of the `CombReorder` algorithm for comb optimization. As the name indicates, the algorithm attempts to permute the input clauses of each comb to sort the comb pivots by their cycle. We assume that combs are double pivot free, i.e., that all pivot variables used along a comb are distinct, and that no pivot variable occurs in the comb's consequent clause. The `CombReorder` algorithm is incorporated into the following high-level algorithm:

Algorithm 6. `ReorderProof`

```
1. Repeat
2.    Eliminate double pivots
3.    Apply CombReorder to all combs
4. Until stable
```

The double pivot elimination step is demonstrated in Figure 5. Note that this procedure may strengthen the consequent clause of the comb from c to $c' \subsetneq c$, which may lead to further simplification of the proof beyond the scope of the comb. The essential property satisfied by the double pivot elimination step is stated in Lemma 3. See [BIFH$^+$08] for a proof and for a full description of the simplification algorithm.

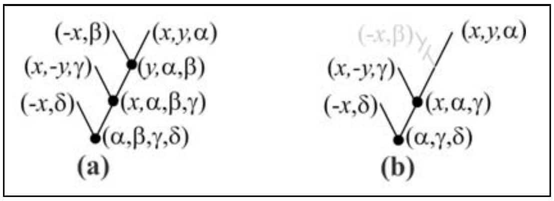


Fig. 5. Double Pivot on x enables resolution trimming

Lemma 3 (Double pivot elimination [BIFH$^+$08]). *Any comb $\overline{C}$ with a double pivot can be converted into a double pivot free comb $\overline{C'}$ where supp$(\overline{C'}) \subsetneq$ supp$(\overline{C})$ and res$(\overline{C'}) \subseteq$ res$(\overline{C})$.*

Our next step is the observation that any algorithm for optimizing a double pivot free comb $\overline{C}$ may restrict its attention to double pivot free reordering of $\overline{C}$. These are the only combs that yield the best possible consequent clause *res*$(\overline{C})$, as stated in the next Lemma:

Lemma 4. *Let $\overline{C}$ be a double pivot free comb. Then: (a) if $\overline{C'}$ is a double pivot free comb with supp$(\overline{C'})$ = supp$(\overline{C})$ then the two combs prove the same result, and (b) the result of any other comb $\overline{C'}$ with supp$(\overline{C'}) \subseteq$ supp$(\overline{C})$ is not stronger or equal to the result of $\overline{C}$, i.e. res$(\overline{C'}) \setminus$ res$(\overline{C})$ is not empty.*

Consider the following non-deterministic reordering algorithm for the given set of clauses C. The algorithm constructs a comb starting from its last resolution.In each iteration it picks a unit clause relative to the current consequent clause c. Theorem 2 states that all legal orders are feasible outputs of the algorithm, and that the choices made by the algorithm can always be completed to a legal comb without any need to backtrack.

Algorithm 7. `CombReorder`

Input: A set of clauses C and a clause c
Output: A valid comb proving c from C with the comb's clauses listed in reverse order

```
1.  While $|C| > 1$
2.     Choose a clause $c' \in C$ such that $c' \setminus c = \{u\}$ for some $u$
3.     Output $c'$
4.     Let $c := c \cup \{-u\}$,  $C := C \setminus c'$
5.  Output the single clause $c'$ remaining in $C$
```

Theorem 2. *Given a double pivot free comb $\overline{C}$, the following properties hold for runs of* `CombReorder` *on input $C = supp(\overline{C})$ and $c = res(\overline{C})$: (a) There is always at least one choice at step 2 regardless of the previous choices made by the algorithm. (b) Any comb generated by the algorithm is a valid double pivot free comb proving c. (c) The* `CombReorder` *algorithm can generate any double pivot free comb with support C.*

The final step needed to make `CombReorder` a deterministic algorithm is a method for choosing the clause c' at step 2. We suggest a greedy rule that chooses c' to maximize $|u|$ under some predetermined variable order, where $|u|$ denotes the variable of the literal u. We suggest two lexicographic orders on the variables, where variables are first ordered by *increasing or decreasing cycle number*, and then by their base, which is given some fixed arbitrary order. The effectiveness of this greedy reordering heuristic is demonstrated both on the synthetic example (see Figure 4) and on real life examples in the next section.

The proofs of Lemma 4 and of Theorem 2 were omitted for lack of space.

5 Experimental Results

We present here four examples taken from the IBM internal benchmark suite, where the proof extension algorithm (Algorithm 1) was able to solve the first three. The proofs were constructed by Mage, IBM's state-of-the-art DPLL-based SAT solver. The following table gives some statistics on the examples and their solution. For each solved example, the fifth column gives the first value of k for which the proof for spec cycle k was extended. As can be seen, for two of the examples extension was done with $\Delta = 2$, which means that one proof was extended to all even spec cycles, and another to all odd cycles. One more interesting fact is that all successful examples were solved by finding a Δ-invariant cut consisting of a single proof node.

Example	FFs	Inputs	Logic elements	Solved at k	Comb order
a	78	94	1128	11	increasing
b	225	174	7252	37 odd, 42 even	decreasing
c	387	170	4913	35 odd, 36 even	decreasing
d	133	89	993	not solved	—

In all the above examples, applying the `CombReorder` algorithm was required for the detection of a Δ-invariant cut. It's interesting to note the significance `CombReorder` has on various proof parameters, where one specific parameter for which visualization is especially effective is the clause width $|n_i.C|$ along the longest comb from init to root. We present in Figure 6 the before and after graph for each of the three examples. In (a), the clause width has decreased and a periodic behavior has emerged. In (b), the improvement in clause width is more dramatic, especially if ignoring the first 200 clauses that can be attributed to an init effect. In (c), the effect is most dramatic from the start.

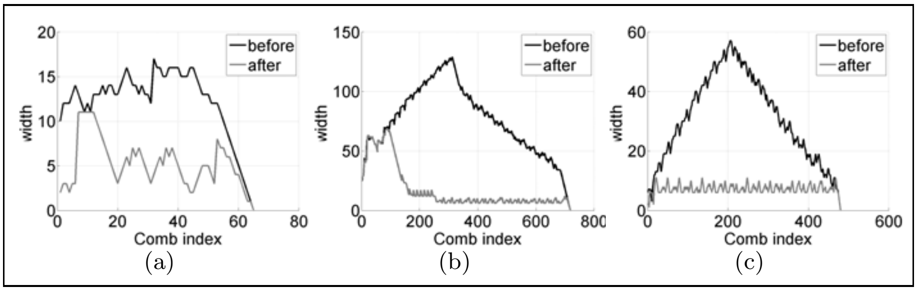


Fig. 6. Width of resolved clause as a function of the clause index

Finally, it's worth noting that the problem with the failed fourth example is the existence of a very wide clause in the proof. For $k = 30$, it consists of 61 literals with cycles ranging from 3 to k. It turns out that this wide clause enters the proof close to the init axioms, and that it takes most of the proof to resolve its literals. The effect of this problem is that Δ-shifted clauses are unlikely to appear later on in the proof, causing Algorithm 4 to fail.

6 Conclusion

This work presents a method for extracting a Δ-invariant cut from a bounded proof of unsatisfiability. We show that such an invariant may be sufficient in extending a bounded proof to an unbounded one. A natural continuation of this work would be to extract weaker invariants to enhance the traditional BMC flow. Effective extraction of invariants relies on the periodic nature of a proof. Thus an essential future component for any method's success is to tune a SAT solver to create cleaner, more structured refutations. A source of optimism is that the natural proof for many real-life problems is periodic by nature.

Acknowledgments. Thanks to Karen Yorav for reviewing the paper drafts, and to Tatyana Veksler and Ofer Strichman for many enlightening talks.

References

[BCCZ99] Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDS. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)

[BIFH$^+$08] Bar-Ilan, O., Fuhrmann, O., Hoory, S., Shacham, O., Strichman, O.: Linear-time reductions of resolution proofs. In: Chockler, H., Hu, A.J. (eds.) HVC 2008. LNCS, vol. 5394, pp. 114–128. Springer, Heidelberg (2008)

[FH09] Fuhrman, O., Hoory, S.: An Example of a Δ-extendible Proof with a Spurious Interpolant. Technical Report H-0265, IBM Haifa Research Lab (2009)

[GKS08] Gershman, R., Koifman, M., Strichman, O.: An approach for extracting a small unsatisfiable core. Form. Methods Syst. Des. 33(1-3), 1–27 (2008)

[JM05] Jhala, R., McMillan, K.L.: Interpolant-based transition relation approximation. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 39–51. Springer, Heidelberg (2005)

[LWS03] Li, B., Wang, C., Somenzi, F.: A satisfiability-based approach to abstraction refinement in model checking. Electr. Notes Theor. Comput. Sci. 89(4) (2003)

[McM03] McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)

[SC85] Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logic. Journal ACM 32, 733–749 (1985)

[SK97] Stoffel, D., Kunz, W.: Record & play: a structural fixed point iteration for sequential circuit verification. In: ICCAD 1997: Proceedings of the 1997 IEEE/ACM international conference on Computer-Aided Design, Washington, DC, USA, pp. 394–399. IEEE Computer Society, Los Alamitos (1997)

[SSS00] Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a sat-solver. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 108–125. Springer, Heidelberg (2000)

[Tse68] Tseitin, G.S.: On the complexity of derivations in the propositional calculus. Studies in Mathematics and Mathematical Logic Part II, 115–125 (1968)

[ZM03a] Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiable boolean formula. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919. Springer, Heidelberg (2004)

[ZM03b] Zhang, L., Malik, S.: Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In: DATE 2003: Proceedings of the conference on Design, Automation and Test in Europe, Washington, DC, USA, p. 10880. IEEE Computer Society, Los Alamitos (2003)