Deriving Small Unsatisfiable Cores with Dominators

Roman Gershman, Maya Koifman, and Ofer Strichman

Technion, Haifa, Israel
ofers@ie.technion.ac.il

Abstract. The problem of finding a small unsatisfiable core of an unsatisfiable CNF formula is addressed. The proposed algorithm, *Trimmer*, iterates over each internal node d in the resolution graph that 'consumes' a large number of clauses M (i.e. a large number of original clauses are present in the unsat core only for proving d) and attempts to prove them without the M clauses. If this is possible, it transforms the resolution graph into a new graph that does not have the M clauses at its core. *Trimmer* can be integrated into a fixpoint framework similarly to Malik and Zhang's fix-point algorithm (RUN_TILL_FIX). We call this option TRIM_TILL_FIX. Experimental evaluation on a large number of industrial CNF unsatisfiable formulas shows that TRIM_TILL_FIX doubles, on average, the number of reduced clauses in comparison to RUN_TILL_FIX. It is also better when used as a component in a bigger system that enforces short timeouts.

1 Introduction

Given an unsatisfiable CNF formula, an unsatisfiable core (UC) is any subset of these clauses that is still unsatisfiable. The problem of finding a *minimum*, *minimal* or just a *small* UC has been addressed rather frequently in the last few years [2,10,16,11,6], partially due to its increasing importance in formal verification.

The decision problem corresponding to finding the *minimum* UC is a Σ_2-complete problem [5] and we are not aware of an algorithms for finding it that scales. Finding a *minimal* UC (any subset of clauses such that the removal of any one of them makes the formula satisfiable), according to Papadimitriou and Wolfe [12], is D^P-complete[1].

It is questionable whether finding a minimal UC has a practical value, however, since a non-minimal UC can be smaller than a minimal one, as long as it is not contained in it. Therefore heuristics that do not guarantee minimality, can be both faster and better than those that guarantee minimality. The latter are useful only when their result is compared to the core from which they started,

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

T. Ball and R.B. Jones (Eds.): CAV 2006, LNCS 4144, pp. 109–122, 2006.

and thus can be used, for example, after another, faster algorithm, has already extracted a small core and cannot find a smaller one.

Typically UCs are needed as part of a larger system (such as an abstraction/refinement loop as we will soon describe), and the influence of the size of the UC on the other parts of the system is only vaguely known. Hence, although more computation time can lead to finding smaller cores, it is not clear whether it is cost-effective in the overall system. This suggests once again that minimality per-se is not so important in practice. Algorithms for extracting small cores should be measured instead by their *velocity*: how many clauses they remove from the initial formula per time unit, on average. They should also be measured by how small they can make the core within a time limit, in comparison with other algorithms, and whether they can contribute to a setting in which several of these algorithms are run sequentially or even in parallel. In Section 6 we measure our suggested technique, called *Trimmer*, with these criteria.

Before we describe previous work on this problem, let us mention some of the typical usages of UCs. A small unsatisfiable core reflects a more precise and focused explanation of the unsatisfiability of a given formula. In verification, it is used in several contexts, some of which are the following. Amla and McMillan [1] suggest to use UCs for a proof-based abstraction-refinement model-checking process: the UC of an unsatisfiable BMC instance contains information on the state variables that are sufficient for proving that no bug can be found up to a given depth; based on these state variables they build a refined abstract model and continue to iterate. Kroening et al. [8] use unsatisfiable cores for an iterative process of solving Presburger formulas: the UC is used for checking whether certain under-approximating restrictions on the solution space were used in the proof of unsatisfiability. If the answer is yes, these restrictions should be relaxed. A similar usage of UCs is by Grumberg et al. [4], in a process of under-approximation and widening of BMC formulas corresponding to a multi-threaded process. Outside verification, the identification of an inconsistent kernel can be important for solving the inconsistency in any constraints satisfaction problem. Further, looking beyond the Propositional world, finding a small unsatisfiable set of constraints is important for the efficiency of decision procedures like MathSat and CVC[15] that rely on explanations of the reason of unsatisfiability in order to prune the search space. The techniques we will discuss in this paper are equally relevant to such systems as they are for systems based on propositional reasoning.

Related Work. Lynce and Silva [10] suggested an approach for finding a minimal UC, in which a new 'clause selector' variable cs_i, $1 \leq i \leq m$, is added to each of the m clauses of the formula (for example, the i^{th} clause $(l_1 \vee l_2)$ is replaced with $(cs_i \vee l_1 \vee l_2)$). The cs variable is set to TRUE iff the clause is not selected. They then use a SAT solver that decides first on the cs variables. If all the clauses become satisfied, it backtracks to the most recent cs variable set to true. If the solver reaches a conflict and consequently backtracks to the cs variables, it means that an unsatisfiable core was found. In such a case it records the size of the core and continues to search for a smaller one, after adding a clause over the cs variables that blocks the solver from repeating the same core. A similar

process was suggested also by Oh et al. [11] (the 'Amuse' algorithm), although they modify the backtracking mechanism so it performs a bottom-up search for a UC instead of searching for a satisfying assignment. Different decision heuristics result in different UCs, which are not necessarily minimal.

Huang suggests the 'MUP' (Minimal Unsatisfiability Prover) algorithm in [6]. Rather than using m clause selector variables, he suggests to augment the clauses with minterms over $\log(m+1)$ variables. The augmented formula, he proves, is minimally unsatisfiable iff there are exactly m models over the y variables (because in this case every clause that is removed makes the formula satisfiable). Hence, the problem of proving that an existing set is minimal is reduced to that of model-counting, which MUP performs with a variable elimination technique over BDDs. This technique can be taken one step further towards finding a minimal core, by running it not more than m times. MUP shows better experimental results than RUN_TILL_FIX (see below), but only, apparently, on hand-made and relatively small formulas, like the pigeonhole problem. None of the benchmarks reported in [6] has more than several thousand clauses, and it is not clear how it scales to industrial problems.

A more practical approach is to find a small core without guaranteeing minimality, while attempting to be efficient and produce intermediate valuable results in case the external process does not wish to wait for the final result. Zhang and Malik [16] were the first in the verification community, as far as we know, to address this problem from a practical point of view. They suggested a simple and effective iterative procedure for deriving a small unsatisfiable core: they extract an unsatisfiable core from an unsatisfiability proof of the formula provided by a SAT solver and then they run the SAT solver again starting from this core, which may result in an even smaller core. Their script RUN_TILL_FIX repeats this process until the core is equal to a core derived in the previous iteration, or, in other words, until it reaches a fixpoint. The solution and its implementation seem to be the most practical one available, and is indeed widely used. The experimental results that we present in Section 6 are compared against RUN_TILL_FIX.

What Is This Article About? We describe a new heuristic, called *Trimmer*, for finding a small UC. *Trimmer* takes the role of zVerify in RUN_TILL_FIX. It can be either applied once (and generate a core smaller or equal to that generated by zVerify) or as part of a fixpoint computation, in an algorithm we call TRIM_TILL_FIX. We will concentrate on *Trimmer* from hereon and return to TRIM_TILL_FIX in the description of the experimental results.

We assume from here on that the reader is familiar with the basic inner-workings of modern DPLL-based SAT solvers, and hence describe those parts of the solver that our algorithm relies on only in general, abstract terms.

New conflict clauses are derived in a process called Conflict Analysis, by (conceptually) traversing backwards the conflict graph and locating the reason for the conflict. This process can be interpreted as a series of resolution steps [16]. The SAT solver can output a graph reflecting the resolution steps, known as the *resolution graph*. The nodes of a resolution graph represent clauses, and the single sink node of this graph represents the empty clause. Each internal node has

two parents, which represent the clauses from which it was resolved. In practice this graph can represent *Hyper-resolution* (a result of several resolution steps) and hence each node can have more than two parents. The general idea of the *Trimmer* algorithm, described in detail in Section 4, is the following. *Trimmer* locates internal nodes in the resolution graph that *dominate* other nodes, called the *minions* (i.e., all the paths from a minion node to the sink node go through the dominator), and checks whether they can be proved without their minions. If the answer is yes, the minions can be removed, and consequently the size of the UC is decreased. In such a case the resolution graph has to be transformed so it reflects the new proof. This transformation is the subject of Section 4.1. *Trimmer* repeats this process until no changes in the graph can be made. Experimental results show that integrating this procedure in a fixpoint script in the style of RUN_TILL_FIX, is better than RUN_TILL_FIX, at least with the relatively short timeouts we tried (30 and 60 minutes). *Trimmer* has the advantage that it generates intermediate results rather fast. Hence, while in many cases RUN_TILL_FIX times out (i.e. it cannot finish the first iteration after the initial core within the time limit), *Trimmer* almost always finishes several iterations by that time, even if in the long run RUN_TILL_FIX produces smaller cores.

2 Preliminaries

Resolution is a proof system for CNF formulas with one inference rule:

$$\frac{(A \vee x)\ (B \vee \neg x)}{(A \vee B)}$$

where A,B are disjunctions of literals (possibly with 0 disjuncts, i.e. the constant FALSE). The clause $(A \vee B)$ is the *resolvent*, and $(A \vee x)$ and $(B \vee \neg x)$ are the *resolving clauses*. The resolvent of the clauses (x) and $(\neg x)$ is the empty clause $(\perp)$. Each application of the resolution rule is called a *resolution step*.

Lemma 1. *A Propositional CNF formula is unsatisfiable if and only if there exists a finite sequence of resolution steps ending with the empty clause.*

A sequence of resolution steps, each one uses the result of the previous step as one of the resolving clauses of the current step, is called *Hyper-resolution*. For example, from

$$(x_1 \vee x_2 \vee x_3)(\neg x_1 \vee x_4)(\neg x_2 \vee x_5)$$

we can derive $(x_3 \vee x_4 \vee x_5)$ by two resolution steps (first over x_1, then over x_2), or by one hyper-resolution step.

The hyper-resolution steps leading to the derivation of the empty clause can be depicted in a *Hyper-resolution graph* (or, simply, a resolution graph). From hereon, we use the terms *node* and *clause* interchangeably, since every node represents a clause.

Definition 1. *A* Hyper-resolution graph *corresponding to an unsatisfiability proof by resolution, is a Directed acyclic Graph $G(V, E, s)$ with a single sink node $s \in V$, in which the nodes represent CNF clauses: the leaf nodes (the sources) represent original clauses, the inner nodes represent clauses derived by resolution, and the sink represents the empty clause. Each node can be inferred from its parent nodes by some sequence of resolution steps.*

Modern DPLL-based SAT solvers can output a Hyper-resolution proof of unsatisfiability. The intermediate clauses in this proof are the conflict clauses that were generated during the run, and that are on a path from the leafs to the empty clause.

We now generalize resolution graphs to *Clause Implication Graphs*:

Definition 2 (Clause Implication Graph). *A* Clause-Implication Graph *(CIG) $G(V, E, s)$ is a directed acyclic graph with a single sink node $s \in V$, in which the nodes represent CNF clauses, and each node is logically implied by the conjunction of clauses represented by its parents.*

A CIG is less restrictive than hyper-resolution graphs. They can have edges such as

- *Subsumption* $((\Phi), (\Phi \vee x))$
- *Reflexive implication* $((\Phi), (\Phi))$
- *Resolution + Subsumption* $((\Phi_1 \vee x), (\Phi_1 \vee \Phi_2 \vee p))$ together with $((\Phi_2 \vee \neg x), (\Phi_1 \vee \Phi_2 \vee p))$

where Φ_1, Φ_2 are disjunctions of literals, and p, x are variables. Other implications forbidden by hyper-resolution are also possible. Figure 1 (left) depicts an example of a Clause Implication Graph.

Let L denote the leaf nodes of a CIG, and assume that s represents the empty clause. By definition of CIG, the conjunction of the L clauses is unsatisfiable, and hence there exists a corresponding resolution proof of unsatisfiability starting from the same nodes. Therefore, for the purpose of finding small UCs, CIGs are sufficient for the analysis. Our construction will begin from the hyper-resolution graph, which can be derived from the resolution trace given to us by the SAT solver, but will transform it to a CIG as the algorithm progresses.

3 Dominators

Prosser [13] introduced the notion of dominance in the context of Flowgraph analysis (originally a term related to code analysis and compilers).

A Flowgraph $G = (V, E, r)$ is a directed graph such that every vertex is reachable from a distinguished root vertex $r \in V$. A vertex $d \in V$ *dominates* $v \in V, v \neq d$, if every path from r to v includes d. d *immediately dominates* v if it dominates v and there is no other node on the path between them that dominates v. We name v a *minion* of d. The set of minions of d is denoted by $M(d)$. A node is called a *dominator* if it dominates at least one node.

In order to adapt the notion of dominators to CIGs, we conceptually reverse the edges of the CIG. Thus, the sink node now becomes the root. Figure 1 (right) presents a *Dominator Tree*, which represents the immediate dominance relation, of a CIG.

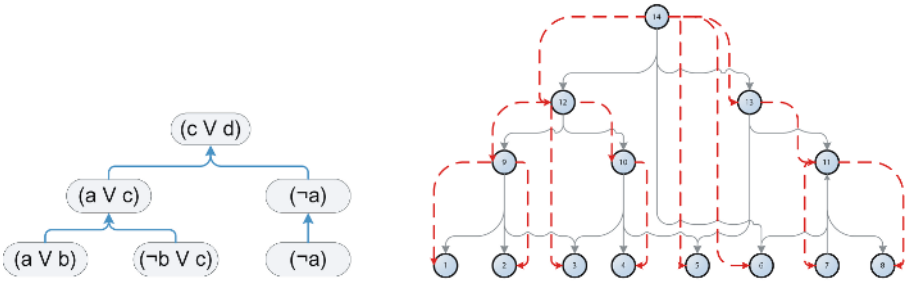


Fig. 1. (Left) A Conflict Implication Graph (CIG) (Right) A Dominator Tree over a reversed CIG. Solid edges belong to the CIG, dashed edges belong to the Dominator Tree. There is a dashed arrow from clause c to c' in Dominator Tree if c is the immediate dominator of c'.

For each vertex in a flowgraph $v \in V$, the set of all vertices dominated by v can be found in polynomial time.

Dominators in a Clause-Implication Graph. We will refer from hereon to a clause set and the formula obtained by conjoining the clauses in the set as the same thing, when the meaning is clear from the context.

Let $LM(d) \subseteq L$ denote the leaf minions of some dominator d. By definition of a CIG, $\bigwedge_{l \in L} l \models s$. The significance of a dominator $d \in V$ in a CIG is that if $L \setminus LM(d) \models d$, then $\bigwedge_{l \in (L \setminus LM(d))} l \models s$. In other words, if d is implied by the leafs which are not its minions, then $LM(d)$ are redundant in the Unsatisfiable Core. Yet removing $LM(d)$ from the CIG is not sufficient, if we want to repeat this process. The problem is that such a removal does not leave us with a valid CIG. The *Trimmer* algorithm, presented in the next section, iterates over dominators in the CIG, and substitutes whenever possible (i.e. when $L \setminus LM(d) \models d$) the old proof of the dominator d with a proof of $L \setminus LM(d) \models d$.

4 The *Trimmer* Algorithm

Our algorithm for decreasing the size of the UC is sketched in Figure 2.

Until Step 5 *Trimmer* is self explanatory. Step 6 Checks whether a dominator d has an alternative proof without $LM(d)$, which amounts to checking the satisfiability of $\varphi' : ((L \setminus LM(d)) \cup \{\neg d\})$, where $\{\neg d\}$ denotes the set of unit clauses corresponding to the negation of the clause d. For example, if $d = (z_1 \vee \ldots \vee z_n)$ is a dominator, then $\{\neg d\}$ are the clauses $(\neg z_1) \ldots (\neg z_n)$, which, for a reason that will soon be clear, we refer to as the *assumptions*. If φ' is satisfiable, the

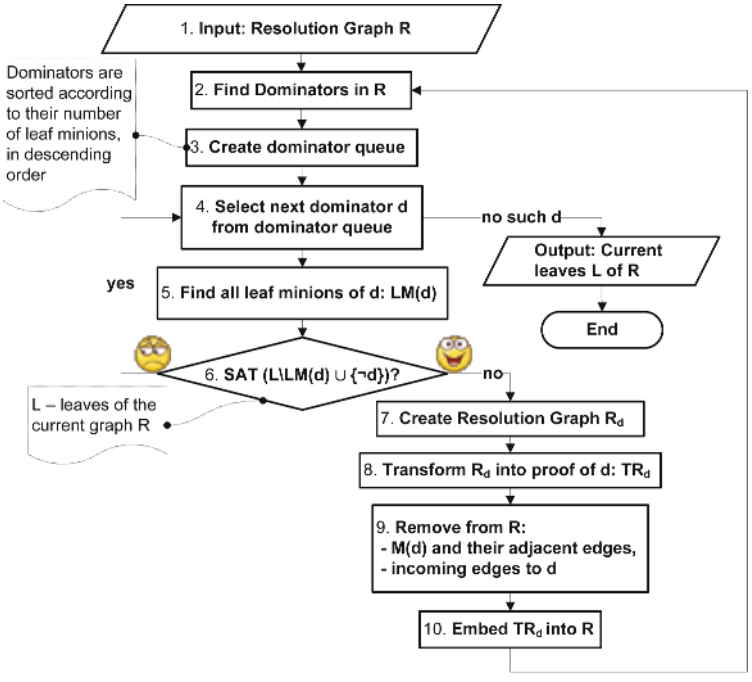


Fig. 2. The *Trimmer* algorithm

attempt failed and it proceeds to the next dominator in the queue. Otherwise, relying on the equivalence

$$((L \setminus LM(d)) \cup \{\neg d\}) \models \bot \qquad \Longleftrightarrow \qquad L \setminus LM(d) \models d,$$

in Step 8 *Trimmer* transforms the hyper-resolution graph R_d into a proof of d, and builds a corresponding CIG TR_d. A transformation is needed because the proof of φ''s unsatisfiability, as generated by the SAT solver, is a proof of the empty clause that uses assumptions. We have to transform it into a proof of d without the assumptions. We discuss two different methods for performing this transformation in Section 4.1. In step 9 *Trimmer* removes from R the graph elements corresponding to the old proof of d and replaces it with the new one, TR_d, in step 10. That is, it removes all the minions of d together with their adjacent edges and incoming edges to d, and *embeds* TR_d into R instead.

Definition 3 (Graph embedding). *The embedding of a graph $G(V, E)$ in a graph $G'(V', E')$, is a graph $G''(V'', E'')$ such that $V'' = V \cup V'$ and $E'' = E \cup E'$.*

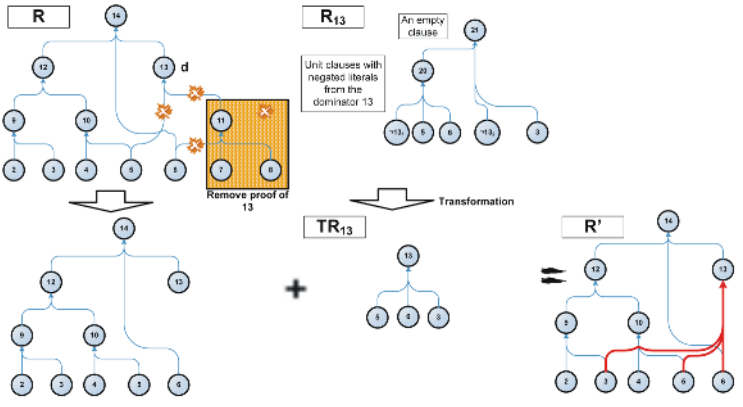
After the old proof is replaced with the new one, the new graph is still a CIG, but has fewer leafs, and hence a smaller unsatisfiable core than the original graph.

4.1 Transforming the Resolution Graph

Recall that in Step 8 *Trimmer* is required to transform the resolution graph R_d, corresponding to a proof of $((L \setminus LM(d)) \cup \{\neg d\}) \models \bot$, into a CIG TR_d that

corresponds to a proof of $L \setminus LM(d) \models d$. We present two possible ways to derive TR_d from R_d. Let $d = (z_1 \vee \ldots \vee z_n)$ be the dominator, and assume that no two literals in this clause are the same. As before we call the unit clauses in $\{\neg d\}$, *assumptions*.

The Simple Transformation. When $((L \setminus LM(d)) \cup \{\neg d\})$ is proven to be unsatisfiable, a subset $L' \subseteq L \setminus LM(d)$ has paths to the empty clause in the resolution graph. This implies that $L' \cup \{\neg d\}$ is unsatisfiable, or equivalently, that L' implies d. Thus, $TR_d(V, E)$ is defined by $V = L' \cup d$ and for all $l' \in L'$, $(l', d) \in E$. Embedding this graph into R corresponds to adding edges from the L' clauses to d itself. The following drawing illustrates a simple transformation and embedding for dominator node 13:



The disadvantage of the simple transformation is that it is too coarse. Since it disregards the conflict clauses, it loses the information about the way these original clauses imply the dominator. Consequently it provides little opportunity for removing more dominators in the main resolution graph. On the other hand, we cannot simply add the conflict clauses, because some of them are derived from the assumptions. What we need is a method for deriving a resolution proof of d from L'. We suggest the *Bubble transformation* method for this derivation.

The Bubble Transformation. For a given clause $d = \{z_1, ..., z_k\}$ and clauses $\{c_1, ..., c_n\}$ we build an assumption set $A = \{(\neg z_1), ..., (\neg z_k)\}$ and a new formula $F = \{c_1, ..., c_n\} \cup A$.

The *Convert* recursive transformation, which appears below, converts a resolution proof Π of the unsatisfiability of F provided by a SAT solver, to a new proof of d. It is initially called with the empty clause. Note that *Convert* is never called with an assumption leaf (these are taken care of in lines 3 and 4), and that the assumption leaves do not participate in the transformed graph. The *Resolve* step resolves between two transformed clauses on the same variable as the original resolution variable, if it still exists in both clauses in different polarity. In

the end of this section we give an intuitive description of an implementation of this procedure, while for now we concentrate on correctness. The relevance of this general procedure to our case is clear: d is the dominator, A is $\{\neg d\}$ and $\{c_1, \ldots, c_n\}$ are the clauses of $L \setminus LM(d)$.

1: **procedure** CONVERT(Node: n)
2: **if** n is leaf **then return** NewNode(n)
3: **if** $left(n) = (\neg z_i)$ **then return** $Convert(right(n))$
4: **if** $right(n) = (\neg z_i)$ **then return** $Convert(left(n))$
5: **return** NewNode($Resolve(Convert(right(n), Convert(left(n))))$)

The following drawing demonstrates a bubble transformation with *Convert*, where $z \in d$:

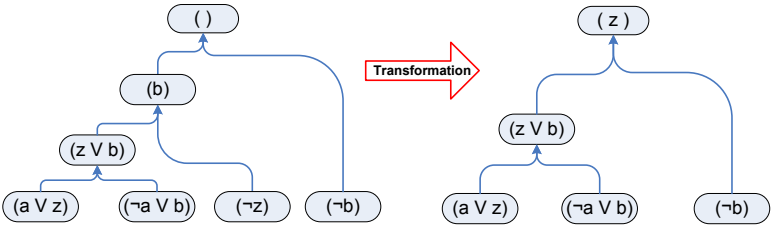
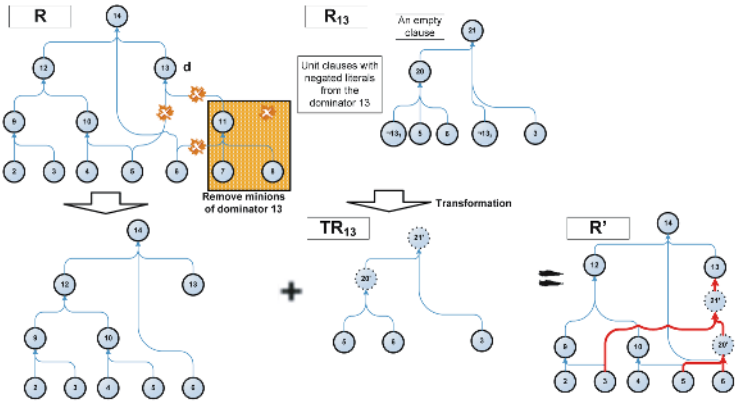


Fig. 3. A bubble proof transformation, where $z \in d$

The following drawing illustrates a bubble transformation and embedding for dominator node 13:



Proposition 1. *Let* $\perp$ *denote the empty clause of the proof* Π *(the proof of* F*'s unsatisfiability). Then* $Convert(\perp)$ *returns a valid resolution proof* Π' *of* $\{c_1, \ldots, c_n\} \models d'$*, s.t.* $literals(d') \subseteq literals(d)$.

Proof. We use the term *proof of unsatisfiability* in order to emphasize that our proof is based on a resolution graph, not a hyper-resolution graph. The information provided by the SAT solver is enough for reconstructing any of these graphs. In order to simplify presentation of the proof even more, we use set notation for clauses to represent their literal sets.

Let $n' = Convert(n)$. We will prove the proposition by induction on the resolution graph structure using the following invariant:

- n' is well-defined
- $n \subseteq n' \subseteq (n \cup d)$.

Base step: if n is a leaf then $n' = n$, which is well-defined and, trivially,

$$n \subseteq n' \subseteq (n \cup d)$$

Induction step: there are two different cases - one for lines 3 and 4, and the other - for line 5.

Lines 3 and 4: Suppose that n is an inner node that was resolved by the two clauses n_l and n_r using the resolution variable t. Let $n'_r = Convert(n_r)$ and $n'_l = Convert(n_l)$. If, w.l.o.g. $n_l = (\neg z_i)$, then, according to the algorithm: (1) $n' = n'_r$. Since the proof is a DAG, n' is well-defined by the induction hypothesis. Also, by induction: (2) $n_r \subseteq n'_r \subseteq (n_r \cup d)$. It must hold that $t = z_i$, since this is the only variable common to n_l and n_r. Therefore: (3) $n \cup \{z_i\} = n_r$. Combining these expressions we get

$$n \overset{(3)}{\subseteq} n_r \overset{(2)}{\subseteq} n'_r \overset{(2)}{\subseteq} (n_r \cup d) \overset{(3)}{=} (n \cup \{z_i\}) \cup d \overset{z_i \in d}{=} (n \cup d)$$

Therefore

$$n \subseteq n'_r \overset{(1)}{=} n' \subseteq (n \cup d)$$

Line 5: Assuming that the invariant holds for n'_r and n'_l, we need to prove that a resolution step is valid on clauses n'_r and n'_l, i.e. that, they have opposite literals of at least one variable. Now, since Π was a valid proof, it must hold that there exists a literal t so that w.l.o.g $t \in n_r$ and $\neg t \in n_l$. Since $n_r \subseteq n'_r$ and $n_l \subseteq n'_l$, it holds that $t \in n'_r$ and $\neg t \in n'_l$. Therefore n' can be derived by resolution between n'_l and n'_r on the same t, and n' is well-defined.

We need to prove that $n \subseteq n' \subseteq (n \cup d)$. Indeed,

$$n \overset{Resolution}{=} ((n_r \cup n_l) \setminus \{t, \neg t\}) \overset{Induction}{\subseteq} ((n'_r \cup n'_l) \setminus \{t, \neg t\}) \overset{Resolution}{=} n'$$

$$\begin{aligned} n' &= ((n'_r \cup n'_l) \setminus \{t, \neg t\}) \overset{Induction}{\subseteq} (((n_r \cup d) \cup (n_l \cup d)) \setminus \{t, \neg t\}) \\ &= (((n_r \cup n_l) \setminus \{t, \neg t\}) \cup (d \setminus \{t, \neg t\})) \;=\; (n \cup (d \setminus \{t, \neg t\})) \;\subseteq\; (n \cup d) \end{aligned}$$

Specially, the invariant implies that for the empty clause $\perp$:

$$Convert(\perp) \;\subseteq\; (\perp \cup d) \;=\; d$$

□

It is easy to show that the resulting graph is a CIG (a resolution graph, actually).

Convert can also be implemented with the following, more intuitive procedure:

1: **for** each assumption $(\neg z_i)$, $1 \leq i \leq n$ in R_d **do**
2: Add z_i to all clauses on all the paths from $(\neg z_i)$ to the sink node.
3: Remove the assumption $(\neg z_i)$ from the graph.

It can be proven that the two procedures are equivalent up to reflexive implications, although this is beyond the scope of this article.

5 Optimizations

Our tool includes the following optimizations.

1. In step 6 of the algorithm (Figure 2) rather than checking $((L \setminus LM(d)) \cup \{\neg d\})$, *Trimmer* conjoins with this formula all the conflict clauses in R that are not on any path from the minions to the sink node. This addition does not change the satisfiability of the formula, because these clauses are logically implied by $L \setminus LM(d)$. But they make the SAT solving stage incremental[14], and hence far more efficient.
2. In step 8, if none of the assumptions participate in the proof, *Trimmer* takes a different route. In this case R_d, which is the proof of unsatisfiability of $((L \setminus LM(d)) \cup \{\neg d\})$, can also be seen as the proof of unsatisfiability of $L \setminus LM(d)$, which are a subset of the clauses in the original formula. Let $L' \subseteq L \setminus LM(d)$ be the leafs of R_d. L' is a UC of $L \setminus LM(d)$, but also of the original formula, and it is smaller than the smallest core known so far (because the core of the current R is L). So, *Trimmer* assigns $R = R_d$ and returns to line 2.

6 Experimental Results

The implementation of the dominator algorithm in our tool TRIMMER is the SLT variant of the Lengauer-Tarjan algorithm[9] (which runs in $O(|E| \log |V|)$ time), as provided by the authors of [3] and published on their web site. We used version 2004.11.15 of zChaff, zVerify and RUN_TILL_FIX for both the comparison and the extraction of the resolution traces.

The benchmark suite is composed of 75 unsatisfiable CNF instances from the industrial category of the SAT competitions in the last two years, from IBM formal verification benchmarks, and BMC instances from the Sun's PicoJava benchmarks that were used in [1]. We did not include benchmarks that timed-out with both TRIMMER and RUN_TILL_FIX. The initial number of clauses ranges from $1,300$ to $800,000$, and the largest initial core size, which is our starting point, has around 160,000 clauses.

We measured two parameters: core reduction (the difference between the final and the initial number of clauses) and average velocity (core reduction divided

by the time spent on the reduction). We used two different timeouts - 1,800 seconds and 3,500 seconds. Since UCs are typically used within a larger system in which they are extracted many times, relatively short timeouts reflect what is practically done for best overall tuning. For such systems velocity seems to be more relevant, assuming the process of decreasing the size of the UC is interrupted after a while, without waiting for the smallest core possible. The timeouts do not include the time of the first run of the solver that extracts the first resolution trace, since this step is common to all tools.

The competing systems in our benchmark are:

- **(Z)** RUN_TILL_FIX.
- **(A)** TRIM_TILL_FIX: running TRIMMER until it terminates, then running zChaff on the new core, then rerunning (T) starting from the new resolution graph, and so on until either a fixpoint or a timeout is reached.
- **(A‖Z)** Running (A) and (Z) in parallel (on different machines) until the first one stops or a timeout is reached. The smallest core produced by the two programs so far is the resulting core of (A‖Z). This approach can be useful if (A) and (Z) are sufficiently different, and neither one dominates the other.
- **(T)** A single run of TRIMMER.

The following table summarizes our results with time out of 3500 sec. *Core reduction* measures the number of clauses removed from the initial core, hence a larger number is better. An intriguing result is the superiority of (A) over (A‖Z) when it comes to clause reduction. This is because the number of clauses counted for (A‖Z) is due to the system that finishes first, which may remove fewer clauses than the other system.

The comparison between (Z) and (A) reveals that TRIM_TILL_FIX removes twice as many clauses on average as RUN_TILL_FIX but RUN_TILL_FIX is 50% faster. Note, however, the medians: the median of TRIM_TILL_FIX is 5 times larger on core reduction and 14 times larger on velocity, which is important in the realm of short timeouts. In other words, if we ran these benchmarks with a shorter timeout, the results would favor TRIM_TILL_FIX much stronger. This is also evident from Figure 5: although (Z)'s velocity is typically better, it suffers from a large number of timeouts, which is counted as 0 velocity in our calculations.

System	Velocity		Core Reduction	
	Med.	Avg.	Med.	Avg.
(Z)	1.1	200.8	729	3126.8
(A)	14.5	130.3	3404	6212.1
(A‖Z)	14.6	239.3	3310	5985.3
(T)	33.0	160.8	1464	3863.1

We also ran a detailed statistical analysis on the results, with the *ordinary sign test* – see [7] for more details. The results, referring to the differences in the

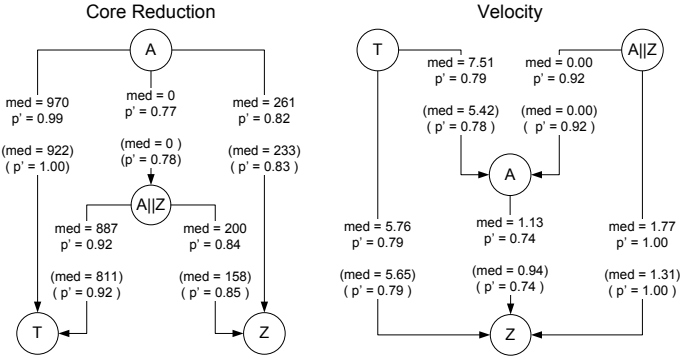


Fig. 4. Results summary of the statistical analysis of the difference in median values of velocity and core reduction. The nodes represent the competing systems, and an edge from a to b represents 99% confidence (i.e. $\alpha = 0.01$) in a's superiority over b. *med* is the median of the difference of values between the parent and its child. p' is the estimated probability of the parent's success (which is equal to the ratio of its success). The results without parentheses correspond to a timeout of $3,500$ sec., and within parentheses to $1,800$ sec. (A) is the ultimate leader in core reduction, and (T) and A||Z are the fastest.

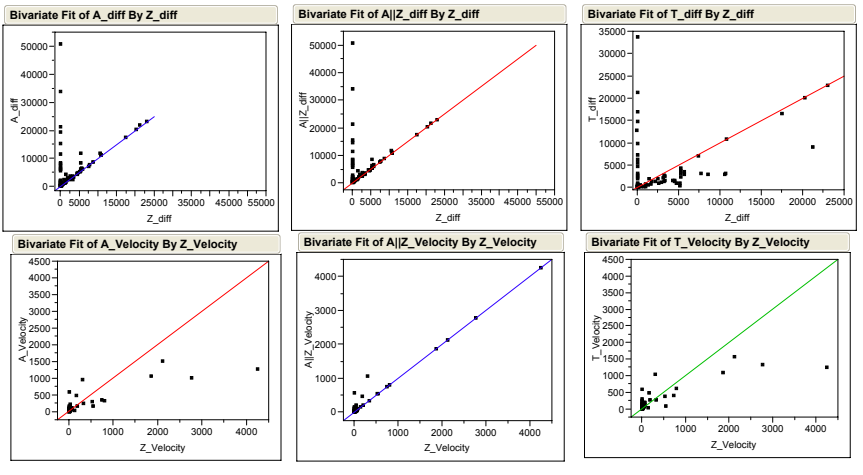


Fig. 5. Core Reduction (top) and Velocity (bottom) of A, A||Z and T Compared to Z

medians of velocity and core reduction, are summarized in Figure 4. We see that there is a statistically significant difference between the competing programs both in velocity and in core reduction, with (A) and (A||Z) being the winners. Note that this result is consistent with our previous conclusions.

As future work we plan to analyze *acceleration*, i.e. the velocity as a function of the elapsed time: this information can lead to new strategies and help choosing the best timeout.

References

1. N. Amla and K. McMillan. Automatic abstraction without counterexamples. In H. Garavel and J. Hatcliff, editors, *TACAS'03*, volume 2619 of *Lect. Notes in Comp. Sci.*, 2003.
2. R. Bruni. Approximating minimal unsatisfiable subformulae by means of adaptive core search. *Discrete Appl. Math.*, 130(2):85–100, 2003.
3. L. Georgiadis, R. F. Werneck, R. E. Tarjan, S. Triantafyllis, and D. I. August. Finding dominators in practice. In *12th Annual European Symposium on Algorithms (ESA 2004)*, volume 3221 of *LNCS*, pages 677–688, 2004.
4. O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation-widening for multi-process systems. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages*, pages 122–131. ACM Press, 2005.
5. A. Gupta. *Learning Abstractions for Model Checking*. PhD thesis, Carnegie Mellon University, 2006. (to be published).
6. J. Huang. Mup: A minimal unsatisfiability prover. In *Proc. of the 10th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 432–437, 2005.
7. M. Koifman. An approach to extracting a small unsatisfiable core. M.sc. thesis, Technion - I.I.T., Israel, Haifa, (to be published) 2006.
8. D. Kroening, J. Ouaknine, S. Seshia, and O. Strichman. Abstraction-based satisfiability solving of Presburger arithmetic. In R. Alur and D. Peled, editors, *Proc.* 16th *Intl. Conference on Computer Aided Verification (CAV'04)*, number 3114 in Lect. Notes in Comp. Sci., pages 308–320, Boston, MA, July 2004. Springer-Verlag.
9. T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.
10. I. Lynce and J. Marques-Silva. On computing minimum unsatisfiable cores. In *Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, pages 305–310, 2004.
11. Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. Amuse: a minimally-unsatisfiable subformula extractor. In *DAC '04*, pages 518–523, 2004.
12. C. H. Papadimitriou and D. Wolfe. The complexity of facets resolved. *J. Comput. Syst. Sci.*, 37(1):2–13, 1988.
13. R. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Proceedings of the Eastern Joint Computer Conference*, pages 133–138, 1959.
14. O. Shtrichman. Prunning techniques for the SAT-based bounded model checking problem. In *proc. of the 11th Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, Edinburgh, Sept. 2001.
15. A. Stump, C. Barrett, and D. Dill. CVC: a cooperating validity checker. In *Proc.* 14th *Intl. Conference on Computer Aided Verification (CAV'02)*, 2002.
16. L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *Theory and Applications of Satisfiability Testing*, 2003.