A Scalable Algorithm for Minimal Unsatisfiable Core Extraction*

Nachum Dershowitz[1], Ziyad Hanna[2], and Alexander Nadel[1,2]

[1] School of Computer Science
Tel Aviv University, Ramat Aviv, Israel
{nachumd, ale1}@post.tau.ac.il
[2] Design Technology Solutions Group
Intel Corporation, Haifa, Israel
{ziyad.hanna, alexander.nadel}@intel.com

Abstract. The task of extracting an unsatisfiable core for a given Boolean formula has been finding more and more applications in recent years. The only existing approach that scales well for large real-world formulas exploits the ability of modern SAT solvers to produce resolution refutations. However, the resulting unsatisfiable cores are suboptimal. We propose a new algorithm for minimal unsatisfiable core extraction, based on a deeper exploration of resolution-refutation properties. Experimental results, confirming that the algorithm is able to find minimal unsatisfiable cores for well-known formal verification benchmarks, are provided.

1 Introduction

Many real-world problems, arising in formal verification of hardware and software, planning and other areas, can be formulated as constraint satisfaction problems, which can be translated into Boolean formulas in conjunctive normal form (CNF). Modern Boolean satisfiability (SAT) solvers, such as Chaff [18, 8] and MiniSAT [7], which implement enhanced versions of the Davis-Putnam-Longeman-Loveland (DPLL) backtrack-search algorithm, are often able to determine whether a large formula is satisfiable or unsatisfiable. When a formula is unsatisfiable, it is often required to find an *unsatisfiable core*—that is, a small unsatisfiable subset of the formula's clauses. Example applications include functional verification of hardware [15], FPGA routing [20], and abstraction refinement [16]. For example, in FPGA routing, an unsatisfiable instance implies that the channel is unroutable. Localizing a small unsatisfiable core is necessary to determine the underlying reasons for the failure. An unsatisfiable core is a *minimal unsatisfiable core (MUC)*, if it becomes satisfiable whenever any one of its clauses is removed. It is always desirable to find a minimal unsatisfiable core, but this problem is very hard (it is D^P-complete; see [22]).

* This research was supported by the Israel Science Foundation (grant no. 250/05). The work of Alexander Nadel was carried out in partial fulfillment of the requirements for a Ph.D.

In this paper, we propose an algorithm that is able to find a minimal unsatisfiable core for large "real-world" formulas. Benchmark families, arising in formal verification of hardware (such as [24]), are of particular interest for us. The only approach for unsatisfiable core extraction that scales well for formal verification benchmarks was independently proposed in [25] and in [10]. We refer to this method as the *EC (Empty-clause Cone)* algorithm. EC exploits the ability of modern SAT solvers to produce a resolution refutation, given an unsatisfiable formula. Most state-of-the-art SAT solvers, beginning with GRASP [14] and including Chaff [18, 8] and MiniSAT [7], implement a DPLL backtrack search enhanced by a failure-driven assertion loop [14]. These solvers explore the assignment tree and create new *conflict clauses* at the leaves of the tree, using resolution on the initial clauses and previously created conflict clauses. This process stops when either a satisfying assignment is found or when the empty clause ($\square$) is derived. In the latter case, SAT solvers are able to produce a *resolution refutation*—a directed acyclic graph (dag), whose vertices are associated with clauses, and whose edges describe resolution relations between clauses. The sources of the refutation are the initial clauses and the empty clause $\square$ is a sink. EC traverses a reversed refutation, starting with $\square$ and taking initial clauses, connected to $\square$, as the unsatisfiable core. Invoking EC until a fixed point is reached [25] allows one to reduce the unsatisfiable core even more. We refer to this algorithm as *EC-fp*. However, the resulting cores can be reduced further.

The basic flow of the algorithm for minimal unsatisfiable core extraction proposed in this paper is composed of the following steps:

1. Produce a resolution refutation Π of a given formula using a SAT solver.
2. Drop from Π all clauses not connected to $\square$. At this point, all the initial clauses remaining in Π comprise an unsatisfiable core.
3. For every initial clause C remaining in Π, check whether it belongs to a minimal unsatisfiable core (MUC) in the following manner:

   Remove C from Π, along with all conflict clauses for which C was required to derive them. Pass all the remaining clauses (including conflict clauses) to a SAT solver.

   - If they are satisfiable, then C belongs to a minimal unsatisfiable core, so continue with another initial clause.
   - If the clauses are unsatisfiable, then C does not belong to a MUC, so replace Π by a new valid resolution refutation not containing C.
4. Terminate when all the initial clauses remaining in Π comprise a MUC.

Related work is discussed in the next section. Section 3 is dedicated to refutation-related definitions. Our basic *Complete Resolution Refutation (CRR)* algorithm is described in Sect. 4, and a pruning technique, enhancing CRR and called *Resolution Refutation-based Pruning (RRP)*, is described in Sect. 5. Experimental results are analyzed in Sect. 6. This is followed up by a brief conclusion.

2 Related Work

Algorithms for unsatisfiable core based on the ability of modern SAT solvers to produce resolution refutations [25, 10] are the most relevant for our purposes for two reasons. First, this approach allows one to deal with real-world examples arising in formal verification. Second, it serves as the basis of our algorithm. We have already described the EC and EC-fp algorithms in the introduction. Here we briefly consider other approaches.

Theoretical work (e.g., [23]) has concentrated on developing efficient algorithms for formulas with small *deficiency* (the number of clauses minus the number of variables). However, real-world formulas have arbitrary (and usually large) deficiency. A number of works considered the harder problem of finding the smallest minimal unsatisfiable core [13, 17], or even finding all minimally unsatisfiable formulas [12]. As one can imagine, these algorithms are not scalable for even moderately large real-world formulas.

In [3, 2], an "adaptive core search" is applied for finding a small unsatisfiable core. The algorithm starts with a very small satisfiable subformula, consisting of *hard* clauses. The unsatisfiable core is built by an iterative process that expands or contracts the current core by a fixed percentage of clauses. The procedure succeeds in finding small, though not necessarily minimal, unsatisfiable cores for the problem instances it was tested on, but these are very small and artificially generated.

Another approach that allows for finding small, but not necessarily minimal, unsatisfiable cores is called AMUSE [21]. In this approach, selector variables are added to each clause and the unsatisfiable core is found by a branch-and-bound algorithm on the updated formula. Selector variables allow the program to implicitly search for unsatisfiable cores using an enhanced version of DPLL on the updated formula. The authors note their methods ability to locate different unsatisfiable cores, as well as its inability to cope with large formulas.

The above described algorithms do not guarantee minimality of the cores extracted. One folk algorithm for minimal unsatisfiable core extraction, which we dub *Naïve*, works as follows: For every clause C in an unsatisfiable formula F, Naïve checks if it belongs to the minimal unsatisfiable core, by invoking a SAT solver on $F \setminus C$. Clause C does not belong to MUC if and only if the solver finds that $F \setminus C$ is unsatisfiable, in which case C is removed from F. In the end, F contains a minimal unsatisfiable core.

The only non-trivial algorithm existing in the current literature that guarantees minimality is MUP [11]. MUP is mainly a prover of minimal unsatisfiability, as opposed to an unsatisfiable core extractor. It decides the minimal unsatisfiability of a CNF formula through BDD manipulation. When MUP is used as a core extractor, it removes one clause at a time until the remaining core is minimal. MUP is able to prove minimal unsatisfiability of some particularly hard classical problems quickly, whereas even just proving unsatisfiability is a challenge for modern SAT solvers. However, the formulas described in [11] are small and arise in areas other than formal verification. We will see in Section 6 that MUP is significantly outperformed by Naïve on formal verification benchmarks.

3 Resolution Refutations

We begin with a resolution refutation of a given unsatisfiable formula, defined as follows:

Definition 1 (Resolution refutation) *Let F be an unsatisfiable CNF formula (set of clauses) and let $\Pi(V, E)$ be a dag whose vertices are clauses.*[3] *Suppose $V = V^i \cup V^c$, where V^i are all the sources of Π, referred to as* initial clauses*, and $V^c = C_1^c, \ldots, C_m^c$ is an ordered set of non-source vertices, referred to as* conflict clauses*. Then, the dag $\Pi(V, E)$ is a* resolution refutation of F *if:*

1. $V^i = F$;
2. *for every conflict clause C_i^c, there exists a resolution derivation $\{D_1, D_2, \ldots, D_k, C_i^c\}$, such that:*
   (a) *for every $j = 1, \ldots, k$, D_j is either an initial clause or a prior conflict clause C_f^c, $f < i$, and*
   (b) *there are edges $D_1 \rightarrow C_i^c, \ldots, D_k \rightarrow C_i^c \in E$ (these are the only edges in E);*
3. *the sink vertex C_m^c is the only empty clause in V.*

For the subsequent discussion, it will be helpful to capture the notion of vertices that are "reachable", or "backward reachable", from a given clause in a given dag.

Definition 2 (Reachable vertices) *Let Π be a dag. A vertex D is* reachable *from C if there is a path (of 0 or more edges) from C to D. The set of all vertices reachable from C in Π is denoted $Re(\Pi, C)$. The set of all vertices unreachable from C in Π is denoted by $\overline{Re}(\Pi, C)$*

Definition 3 (Backward reachable vertices) *Let Π be a dag. A vertex D is* backward reachable *from C if there is a path (of 0 or more edges) from D to C. The set of all vertices backward reachable from C in Π is denoted by $BRe(\Pi, C)$. The set of all vertices not backward reachable from C in Π is denoted $\overline{BRe}(\Pi, C)$.*

For example, consider the resolution refutation of Fig. 1. We have $Re(\Pi, C_5^i) = \{C_5^i, C_2^c, C_3^c, C_4^c, C_5^c\}$ and $BRe(\Pi, C_4^c) = \{C_4^c, C_5^i, C_6^i\}$.

Resolution refutations trace all resolution derivations of conflict clauses, including the empty clause. Generally, not all clauses of a refutation are required to derive $\square$, but only such that are backward reachable from $\square$. It is not hard to see that even if all other clauses and related edges are omitted, the remaining graph is still a refutation. We refer to such refutations as *non-redundant* (see Definition 4). The refutation in Fig. 1 is non-redundant.

To retrieve a non-redundant subgraph of a refutation, it is sufficient to take $BRe(\Pi, \square)$ as the vertex set and to restrict the edge set E to edges having both ends in $BRe(\Pi, \square)$. We denote a non-redundant subgraph of a refutation Π by $\Pi \restriction_{BRe(\Pi, \square)}$. Observe that $\Pi \restriction_{BRe(\Pi, \square)}$ is a valid non-redundant refutation.

[3] From now on, we shall use the terms "vertex" and "clause" interchangeably in the context of resolution refutation.

Definition 4 (Non-redundant resolution refutation) *A resolution refutation Π is* non-redundant *if there is a path in Π from every clause to $\square$.*

Lastly, we define a relative hardness of a resolution refutation.

Definition 5 (Relative hardness) *The* relative hardness *of a resolution refutation is the ratio between the total number of clauses and the number of initial clauses.*

4 The Complete Resolution Refutation (CRR) Algorithm

Our goal is to find a minimal unsatisfiable core of a given unsatisfiable formula F. The proposed *CRR* method is displayed as Algorithm 1.

Algorithm 1 (CRR). Returns a MUC, given an unsatisfiable formula F.

Build a non-redundant refutation $\Pi(V^i \cup V^c, E)$
while unmarked clauses exist in V^i **do**
  $C \leftarrow$ *PickUnmarkedClause*(V^i)
  Invoke a SAT solver on $\overline{Re}(\Pi, C)$
  if $\overline{Re}(\Pi, C)$ is *satisfiable* **then**
    Mark C as a MUC member
  else
    Let $G = \overline{Re}(\Pi, C)$
    Build resolution refutation $\Pi'(V^i_G \cup V^c_G, E_G)$
    $V^i \leftarrow V^i \setminus \{C\}$
    $V^c \leftarrow (V^c \setminus Re(\Pi, C)) \cup V^c_G$
    $E \leftarrow (E \setminus Re^E(\Pi, C)) \cup E_G$
    $\Pi(V^i \cup V^c, E) \leftarrow \Pi(V^i \cup V^c, E) \restriction_{BRe(\Pi,\square)}$
return V^i

First, CRR builds a non-redundant resolution refutation. Invoking a SAT solver for constructing a (possibly redundant) resolution refutation $\Pi(V, E)$ and restricting it to $\Pi \restriction_{BRe(\Pi,\square)}$ is sufficient for this purpose.

Suppose $\Pi(V^i \cup V^c, E)$ is a non-redundant refutation. CRR checks, for every unmarked clause C left in V^i, whether C belongs to the minimal unsatisfiable core. Initially, all clauses are unmarked. At each stage of the algorithm, CRR maintains a valid refutation of F.

Recall from Definition 2 that $\overline{Re}(\Pi, C)$ is the set of all vertices in Π unreachable from C. By construction of Π, the $\overline{Re}(\Pi, C)$ clauses were derived independently of C. To check whether C belongs to the minimal unsatisfiable core, we provide the SAT solver with $\overline{Re}(\Pi, C)$, including the conflict clauses. We are trying to *complete the resolution refutation*, not using C as one of the sources. Observe that $\square$ is always reachable from C, since Π is a non-redundant refutation; thus $\square$ is never passed as an input to the SAT solver. We let the SAT

solver try to derive $\square$, using $\overline{Re}(\Pi, C)$ as the input formula, or else prove that $\overline{Re}(\Pi, C)$ is satisfiable.

In the latter case, we conclude that C must belong to the minimal unsatisfiable core, since we found a model for an unsatisfiable subset of initial clauses minus C. Hence, if the SAT solver returns *satisfiable*, the algorithm marks C (line 6) and moves to the next initial clause. However, if the SAT solver returns *unsatisfiable*, we cannot simply remove C from F and move to the next clause, since we need to keep a valid resolution refutation for our algorithm to work properly. We describe the construction of a valid refutation (lines 8–13) next.

Let $G = \overline{Re}(\Pi, C)$. The SAT solver produces a new resolution refutation $\Pi'(V_G^i \cup V_G^c, E_G)$ for G, whose sources are the clauses $\overline{Re}(\Pi, C)$. We cannot use Π' as the refutation for the subsequent iterations, since the sources of the refutation may only be initial clauses of F. However, the "superfluous" sources of Π' are conflict clauses of Π, unreachable from C, and thus are derivable from $V^i \setminus C$ using resolution relations, corresponding to edges of Π. Hence, it is sufficient to augment Π' with such edges of Π that connect $V^i \setminus C$ and $\overline{Re}(\Pi, C)$ to obtain a valid refutation whose initial clauses belong to F. Algorithm CRR constructs a new refutation, whose sources are $V^i \setminus C$; the conflict clauses are $\overline{Re}(\Pi, C) \cup V_G^c$ and the edges are $(E \setminus (V_1, V_2) | (V_1 \in Re(\Pi, C)\ or\ V_2 \in Re(\Pi, C))) \cup E_G$. This new refutation might be redundant, since $\Pi'(V_G^i \cup V_G^c, E_G)$ is not guaranteed to be non-redundant. Therefore, prior to checking the next clause, we reduce the new refutation to a non-redundant one. Observe that in the process of reduction to a non-redundant subgraph, some of the initial clauses of F, may be omitted; hence, each time a clause C is found not to belong to the minimal unsatisfiable core, we potentially drop not only C, but also other clauses.

We demonstrate the process of completing a refutation on the example from Fig. 1. Suppose we are checking whether C_1^i belongs to the minimal unsatisfiable core. In this case, $G = \overline{Re}(\Pi, C_1^i) = \{C_2^i, C_3^i, C_4^i, C_5^i, C_6^i, C_7^i, C_2^c, C_4^c\}$. The SAT solver receives G as the input formula. It is not hard to check that G is unsatisfiable. One refutation of G is $\Pi'(V_G^i \cup V_G^c, E_G)$, where $V_G^i = \{C_2^i, C_2^c, C_7^i, C_4^c\}$, $V_G^c = (D_1 = \square, D_2 = a \vee b)$, and $E_G = \{(C_2^i, D_2), (C_2^c, D_2), (D_2, D_1), (C_7^i, D_1), (C_4^c, D_1)\}$. Therefore, C_1^i, C_1^c, C_3^c, C_5^c and related edges are excluded from the refutation of F, whereas D_2, D_1 and related edges are added to the refutation of F. In this case, the resulting refutation is non-redundant.

We did not define how the function *PickUnmarkedClause* should pick clauses (line 3). Our current implementation picks clauses in the order in which clauses appear in the given formula. Development of sophisticated heuristics is left for future research.

Another direction that may lead to a speed-up of CRR is adjusting the SAT solver for the purposes of CRR algorithm, considering that the SAT solver is invoked thousands of times on rather easy instances. Integrating the data structures of CRR and the SAT solver, tuning SAT solver's heuristics for CRR, and holding the refutation in-memory, rather than on disk (as suggested in [25] for EC), can be helpful.

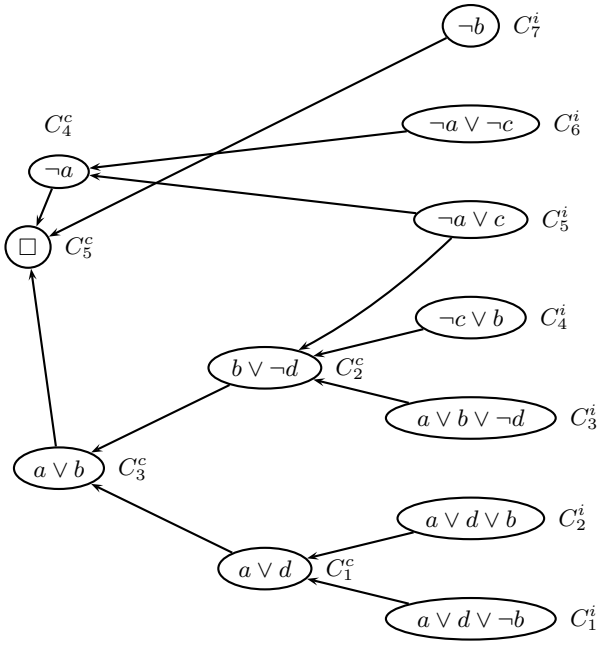

Fig. 1. Resolution refutation example

5 Resolution-Refutation-Based Pruning

In this section, we propose an enhancement of Algorithm CRR by developing resolution refutation-based pruning techniques for when a SAT solver invoked on $\overline{Re}(\Pi, C)$ to check whether it is possible to complete a refutation without C. We refer to the pruning technique, proposed in this section, *Resolution Refutation-based Pruning (RRP)*. We presume that the reader is familiar with the functionality of a modern SAT solver. (An overview is given in [19].)

An assignment σ *falsifies* a clause C, if every literal of C is *false* under σ. An assignment σ *falsifies* a set of clauses P if every clause $C \in P$ is falsified by σ. We claim that a model for $\overline{Re}(\Pi, C)$ can only be found under such a partial assignment that falsifies every clause in some path from C to the empty clause in $Re(\Pi, C)$. The intuitive reason is that every other partial assignment satisfies C and must falsify $\overline{Re}(\Pi, C)$, since F is unsatisfiable. A formal statement and proof is provided in Theorem 1 below.

Consider the example of Fig. 1. Suppose the currently visited clause is C_5^i. There exist two paths from C_5^i to the empty clause C_5^c, namely $\{C_5^i, C_4^c, C_5^c\}$ and $\{C_5^i, C_2^c, C_3^c, C_5^c\}$. A model for $\overline{Re}(\Pi, C_5^i)$ can only be found in a subspace under the partial assignment $\{a = 1, c = 0\}$, falsifying all the clauses of the first path. The clauses of the second path cannot be falsified, since a must be *true* to falsify clause C_5^i and *false* to falsify clause C_3^c.

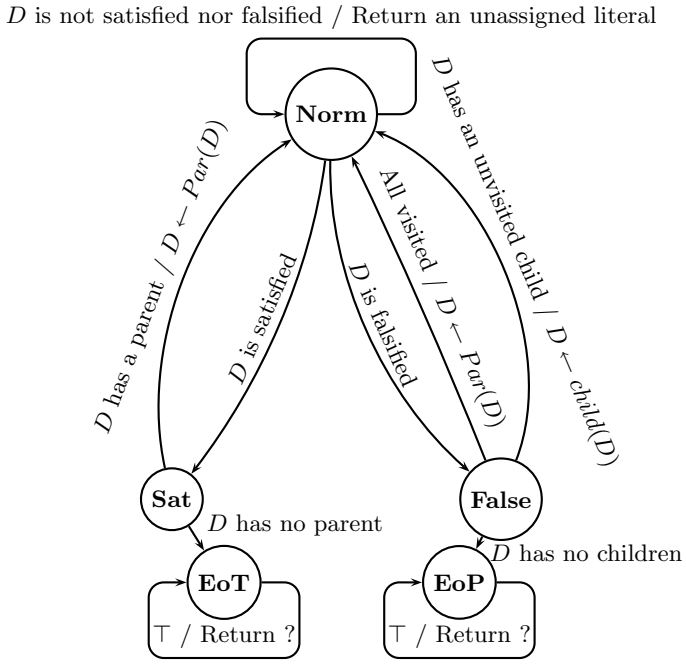

Fig. 2. Function RRP_Decide, represented as a transition relation. This function is invoked by the decision engine of a SAT solver, implementing the RRP pruning technique.

Denote a subtree connecting C and $\square$ by $\Pi\restriction_C$. The proposed pruning technique, RRP, is integrated into the decision engine of the SAT solver. The solver receives $\Pi\restriction_C$, together with the input formula $\overline{Re}(\Pi, C)$. The decision engine of the SAT solver explores $\Pi\restriction_C$ in a depth-first manner, picking unassigned variables in the currently explored path as decision variables and assigning them *false*. As usual, Boolean Constraint Propagation (BCP) follows each assignment. Backtracking in $\Pi\restriction_C$ is tightly related to backtracking in the assignment space. Both happen when a satisfied clause in $\Pi\restriction_C$ is found or when a new conflict clause is discovered during BCP. After a particular path in $\Pi\restriction_C$ has been falsified, a general-purpose decision heuristic is used until the SAT solver either finds a satisfying assignment or proves that no such assignment can be found under the currently explored path. This process continues until either a model is found or the decision engine has completed exploring $\Pi\restriction_C$. In the latter case, one can be sure that no model for $\overline{Re}(\Pi, C)$ exists. However, the SAT solver should continue its work to produce a refutation.

We need to describe in greater detail the changes in the decision and conflict analysis engines of the SAT solver required to implement RRP. The decision engine first invokes function *RRP_Decide*, depicted in Fig. 2, as a state transition relation. Each transition edge has a label consisting of a condition under which

the state transition occurs and an operation, executed upon transition. The state can be one of the following:

(**Norm**)	normal;
(**Sat**)	the currently explored clause is satisfied;
(**False**)	the currently explored clause is falsified;
(**EoT**)	subgraph $\Pi\restriction_C$ has been explored;
(**EoF**)	all clauses in the currently explored path are falsified.

The states are managed globally, that is, if *RRP_Decide* moves to state S, it will start in state S when invoked next. A pointer D to the currently visited clause of $\Pi\restriction_C$ is also managed globally. The state transition relation is initialized prior to the first invocation of the decision engine. Pointer D is initialized to C and the initial state is **Norm**.

State **Norm** corresponds to a situation when the algorithm does not know what the status of D is. If D is neither satisfied nor falsified, *RRP_Decide* returns a negation of some literal of D, which will serve as the next decision variable. If D is satisfied, the algorithm moves to **Sat**. Observe that a clause may become satisfied only as a result of BCP. Encountering a satisfied clause means that the currently explored path cannot be falsified, and we can backtrack. Suppose we are in **Sat**, meaning that D is satisfied. If D has a parent, the algorithm backtracks by moving D to point to its parent, and goes back to **Norm**; otherwise, the tree is explored and the algorithm moves to **EoT**. In this case, *RRP_Decide* returns an unknown value and a general-purpose heuristic must be used. Consider now the case when the state is **Norm** and D is falsified. The algorithm moves to **False**. Here, one of the three conditions hold:

(a) D has an unvisited child S. In this case D is updated to point to S and *RRP_Decide* moves back to **Norm**.
(b) All children of D are visited. In this case, we backtrack by moving D back to its parent and go back to **Norm**.
(c) D has no children. In this case, all the clauses in the currently explored path are falsified. The algorithm moves to **EoF**; *RRP_Decide* returns an unknown value; and a general-purpose heuristic must be used.

To complete the picture, we describe the changes to the conflict analysis engine required to implement RRP. One of the main tasks of conflict analysis in modern SAT solvers is to decide to which level in the decision tree the algorithm should backtrack. Let this decision level be bl. When invoked in RRP mode, the conflict analysis engine must also find whether it is required to backtrack in $\Pi\restriction_C$, and to which clause. The goal is to backtrack to the highest clause B in the currently explored path in $\Pi\restriction_C$, such that B has unassigned literals. Recall that D is a pointer to the currently visited clause of $\Pi\restriction_C$. Denote by $mdl(D)$ the maximal decision level of D's literals. If $bl \geq mdl(D)$, the algorithm does nothing; otherwise, it finds the first predecessor of D in $\Pi\restriction_C$, such that $bl < mdl(B)$ and sets $D \leftarrow B$.

We found experimentally that the optimal performance for RRP is achieved when it explores $\Pi\restriction_C$ starting from $\square$ toward C (and not vice-versa). In other

words, prior to the search, the SAT solver reverses all the edges of $\Pi\restriction_C$ and sets the pointer D to $\Box$, rather than to C. (By default, the current version of RRP explores the graph only until a predefined depth of 50.) The next literal from the currently visited clause is chosen by preferring an unassigned literal with the maximal number of appearances in recent conflict clause derivations (similar to Berkmin's [9] heuristic for SAT). The next visited child is chosen arbitrary. Further tuning of the algorithm is left for future research.

Theorem 1 *Let $\Pi(V^i, V^c)$ be a non-redundant resolution refutation. Let $C \in V^i$ be an initial clause and σ be an assignment. Then, if $\sigma \models \overline{Re}(\Pi, C)$, there is a path $P = \{C, \ldots, C^c_m\}$ in $Re(\Pi, C)$, connecting C to the empty clause[4], such that σ falsifies every clause in P.*

Proof. Suppose, on the contrary, that no such path exists. Then, there exists a satisfiable vertex cut U in Π. But the empty clause is derivable from U, since it is a vertex cut; thus U unsatisfiable, a contradiction.

6 Experimental Results

We have implemented CRR and RRP in the framework of the VE solver. VE, a variant of the industrial solver Eureka, is a modern SAT solver, which implements the following state-of-the-art algorithms and techniques for SAT: it uses 1UIP conflict clause recording [18], enhanced by conflict clause minimization [6], frequent search restarts [9, 8], an aggressive clause deletion strategy [9, 8], and decision stack shrinking [19, 8]. VE uses Berkmin's decision heuristic [9] until 4000 conflicts are detected, and then switches to the CBH heuristic, described in [5]. We used benchmarks from four well-known unsatisfiable families, taken from bounded model checking (*barrel*, *longmult*) [1] and microprocessor verification (*fvp-unsat.2.0*, *pipe_unsat_1.0*) [24]. All the instances we used appear in the first column of Table 1. The experiments on families *barrel* and *fvp-unsat.2.0* were carried out on a machine with 4Gb of memory and two Intel Xeon CPU 3.06 processors. A machine with the same amount of memory and two Intel Xeon CPU 3.20 processors was used for experiments with the families *longmult* and *pipe_unsat_1.0*.

The main goal of the experiments was to compare the algorithms for MUC extraction proposed in this paper with existing algorithms. Two algorithms for MUC extraction exist in the literature: One is the folk algorithm Naïve; the other is BDD-based MUP [11]. In preliminary experiments, we found that Naïve demonstrates its best performance on formulas that are first trimmed down by a suboptimal algorithm for unsatisfiable core extraction. We tried Naïve in combination with EC, EC-fp and AMUSE and found that EC-fp is the best front-end for Naïve. In our main experiments, we used Naïve, combined with EC-fp, and Naïve combined with AMUSE. We have also found that MUP demonstrates

[4] The empty clause always belongs to $Re(\Pi, C)$, since $\Pi(V^i, V^c)$ is non-redundant.

Table 1. Comparing MUC algorithms. Columns *Instance*, *Var* and *Cls* contain instance name, number of variables, and clauses. The next five columns contain execution times of respective algorithms. The cut-off time is 24 hours (86400 seconds). Column *R. R. Hrd.* contains the relative hardness of the final resolution refutation, produced by CRR+RRP. The bold values are the best.

			CRR		**Naïve**		**MUP**	**R.R.**
Instance	**Var**	**Cls**	**RRP**	**plain**	**EC-fp**	**AMUSE**	**EC-fp**	**Hrd.**
2pipe	892	6695	34	**31**	127	198	>86400	1.6
2pipe_1_ooo	834	7026	**20**	**20**	131	186	>86400	1.5
2pipe_2_ooo	925	8213	40	**39**	152	288	45050	1.6
3pipe	2468	27533	**572**	646	2180	83124	>86400	1.5
3pipe_1_ooo	2223	26561	**431**	678	1941	7199	>86400	1.5
3pipe_2_ooo	2400	29981	**603**	837	3298	54275	>86400	1.6
3pipe_3_ooo	2577	33270	**636**	1021	4236	>86400	mem-out	1.5
4pipe	4237	80213	**3527**	4933	24111	>86400	>86400	1.4
4pipe_1_ooo	4647	74554	**4414**	10944	25074	>86400	mem-out	1.7
4pipe_2_ooo	4941	82207	**5190**	12284	49609	>86400	mem-out	1.7
4pipe_3_ooo	5233	89473	**6159**	15867	41199	>86400	mem-out	1.6
4pipe_4_ooo	5525	96480	**6369**	16317	47394	>86400	mem-out	1.6
2pipe_k	860	6693	26	**25**	233	256	6174	1.5
3pipe_k	2391	27405	**411**	493	2147	12544	mem-out	1.5
4pipe_k	5095	79489	**3112**	3651	15112	>86400	>86400	1.5
5pipe_k	9330	189109	**13836**	17910	83402	>86400	mem-out	1.4
barrel3	275	942	**1**	**1**	9	13	15	2
barrel4	578	2035	**6**	**6**	34	47	980	1.8
barrel5	1407	5383	93	**86**	406	326	mem-out	1.8
barrel6	2306	8931	**351**	423	4099	4173	mem-out	1.8
barrel7	3523	13765	**970**	1155	6213	24875	mem-out	1.9
barrel8	5106	20083	**2509**	2859	>86400	>86400	mem-out	1.8
longmult1	791	2335	**0**	**0**	10	8	1	1.6
longmult2	1164	3525	**0**	1	16	22	1	1.8
longmult3	1555	4767	**2**	**2**	56	64	5	2.1
longmult4	1966	6069	8	**7**	109	152	13	2.6
longmult5	2397	7431	74	**31**	196	463	35	3.6
longmult6	2848	8853	**288**	311	749	2911	5084	5.6
longmult7	3319	10335	6217	**3076**	6154	32791	68016	14.2

its best performance when combined with EC-fp, while CRR performs the best when the first refutation is constructed by EC, rather than EC-fp. Consequently, we provide results for MUP combined with EC-fp and CRR combined with EC. MUP requires a so-called "decomposition tree", in addition to the CNF formula. We used the c2d package [4] for decomposition tree construction.

Table 1 summarizes the results of a comparison of the performance of five algorithms for minimal unsatisfiable core extraction. The first is the CRR algorithm, enhanced by RRP; the second is the plain CRR algorithm. Both use EC as front-end. The third and fourth are the Naïve algorithm, combined with EC-fp and AMUSE, respectively. The last is MUP, combined with EC-fp. We do not provide the sizes of the resulted unsatisfiable cores due to space limitations. We found that the sizes do not vary much between different algorithms.

One can see that the BDD-based MUP can compete with algorithms based on SAT only on the 5 easiest instances of family *longmult*. MUP does not succeed in solving most of the other instances due to time-out or memory-out. Observe also that EC-fp performs much better than AMUSE when used as a front-end for Naïve. Thus, we will use the combination of EC-fp and Naïve as the baseline for comparison of existing algorithms with those proposed in this paper.

First, we compare CRR+RRP and Naïve. CRR+RRP shows a consistent speed-up of 3.7–9.6X over Naïve on every instance from the *fvp-unsat.2.0* and *pipe_unsat_1.0* families. The performance boost is even more significant for the *barrel* family. It is at least as large as 34.4X for the hardest *barrel* instance. CRR+RRP outperforms Naïve on the first 6 instances of *longmult* family; however, it is slightly slower on the hardest instances of *longmult* family.

Next, we analyze the impact of RRP. RRP improves the performance on every instance of *pipe_unsat_1.0*, expect the easiest one. It also improves the performance on the 9 hardest instances of the *fvp-unsat.2.0* family and on the 3 hardest instances of *barrel*. The most significant speed-up of RRP is about 2.5X, achieved on hard instances of family *fvp-unsat.2.0*. The only family for which RRP is usually unhelpful is *longmult*.

Overall, CRR outperforms Naïve on every benchmark, whereas CRR+RRP outperforms Naïve on 28 out of 29 benchmarks. This demonstrates that our algorithms are justified practically. Usually, the speed-up of these algorithms over Naïve varies between 4 to 10X, but it can be as large as 34X (for the hardest instance of *barrel* family) and as small as 2X (for the hardest instance of *longmult*).

A natural question is why the complex instances of family *longmult* are hard for CRR, and even harder for RRP. The key difference between *longmult* and other families is the hardness of the resolution proof. The relative hardness of a resolution refutation produced by CRR+RRP varies between 1.4 to 2 for every instance of every family, except *longmult*, where it reaches 14.2 for the *longmult7* instance. When the refutation is too complex, the exploration of $\overline{Re}(\Pi, C)$ executed by RRP is too complicated; thus, plain CRR is advantageous over CRR+RRP. Also, when the refutation is too complex, it is costly to perform

traversal operations, as required by CRR. This explains why the advantage of CRR over Naïve is as small as 2X.

7 Conclusions

We have proposed an algorithm for minimal unsatisfiable core extraction. It builds a resolution refutation using a SAT solver and finds a first approximation of a minimal unsatisfiable core. Then it checks, for every remaining initial clause C, if it belongs to the minimal unsatisfiable core. The algorithm reuses conflict clauses and resolution relations throughout its execution. We have demonstrated that our algorithm is faster than currently existing algorithms by a factor of 6 or more on large problems with non-overly hard resolution proofs, and that it can find minimal unsatisfiable cores for real-world formal verification benchmarks.

Acknowledgments

We thank Jinbo Huang for his help in obtaining and using MUP and Zaher Andraus for his help in receiving AMUSE.

References

1. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, 1999.
2. R. Bruni. Approximating minimal unsatisfiable subformulae by means of adaptive core search. *Discrete Applied Mathematics*, 130(2):85–100, 2003.
3. R. Bruni and A. Sassano. Restoring Satisfiability or Maintaining Unsatisfiability by Finding Small Unsatisfiable Subformulae. In *Proceedings of the Workshop on Theory and Application of Satisfiability Testing (SAT'01)*, 2001.
4. A. Darwiche. New advances in compiling CNF into decomposable negation normal form. In *Proceedings of the 16th European Conference on Artificial Intelligence, (ECAI'04)*, 2004.
5. N. Dershowitz, Z. Hanna, and A. Nadel. A clause-based heuristic for SAT solvers. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, 2005.
6. N. Eén and N. Sörensson. MiniSat v1.13 — a SAT solver with conflict-clause minimization. Available at `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/cgi/MiniSat_v1.13_short.ps.gz.cgi`, 2002.
7. N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, 2003.
8. Z. Fu, Y. Mahajan, and S. Malik. ZChaff2004: an efficient SAT solver. In *Proceedings of Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, 2004.
9. E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation, and Test in Europe (DATE '02)*, 2002.

10. E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, 2003.
11. J. Huang. MUP: A minimal unsatisfiability prover. In *Proceedings of the Tenth Asia and South Pacific Design Automation Conference (ASP-DAC'05)*, 2005.
12. M. H. Liffton and K. A. Sakallah. On finding all minimally unsatisfiable subformulas. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, 2005.
13. I. Lynce and J. P. Marques-Silva. On computing minimum unsatisfiable cores. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, 2004.
14. J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
15. K. L. McMillan. *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, Pittsburgh, PA, USA, 1992.
16. K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, 2003.
17. M. N. Mneimneh, I. Lynce, Z. S. Andraus, J. P. Marques-Silva, and K. A. Sakallah. A branch and bound algorithm for extracting smallest minimal unsatisfiable formulas. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, 2005.
18. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.
19. A. Nadel. Backtrack search algorithms for propositional logic satisfiability: Review and innovations. Master's thesis, Hebrew Univeristy of Jerusalem, Jerusalem, Israel, November 2002.
20. G.-J. Nam, F. Aloul, K. Sakallah, and R. Rutenbar. A comparative study of two Boolean formulations of FPGA detailed routing constraints. In *Proceedings of the 2001 International Symposium on Physical Design (ISPD'01)*, 2001.
21. Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. AMUSE: a minimally-unsatisfiable subformula extractor. In *Proceedings of the 41th Design Automation Conference (DAC'04)*, 2004.
22. C. H. Papadimitriou and M. Yannakakis. The complexity of facets (and some facets of complexity). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing (STOC'82)*, 1982.
23. S. Szeider. Minimal unsatisfiable formulas with bounded clause-variable difference are fixed-parameter tractable. *Journal of Computer and System Sciences*, 69(4):656–674, 2004.
24. M. Velev and R. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.
25. L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In *Proceedings of Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, 2003.