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. We propose a new algorithm for minimal unsatisfiable core extraction, based on a deeper exploration of resolution-refutation properties. We provide experimental results on formal verification benchmarks confirming that our algorithm finds smaller cores than suboptimal algorithms; and that it runs faster than those algorithms that guarantee minimality of the core. (A more complete version of this paper may be found at arXiv.org/pdf/cs.LO/0605085.)

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). 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, field-programmable gate-array (FPGA) routing, and abstraction refinement. An unsatisfiable core is a *minimal unsatisfiable core (MUC)*, if it becomes satisfiable whenever any one of its clauses is removed.

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 [8]), are of particular interest to us.

The folk algorithm for MUC 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 core by invoking a propositional satisfiability (SAT) solver on F, but without clause C. Clause C does not belong to a minimal core 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.

There are four more practical approaches for unsatisfiable core extraction in the current literature: adaptive core search [2], AMUSE [7], MUP [5] and a

* We thank Jinbo Huang and Zaher Andraus for their help in providing MUP and AMUSE, respectively. The work of Alexander Nadel was carried out in partial fulfillment of the requirements for a Ph.D. This research was supported in part by the Israel Science Foundation (grant no. 250/05).

resolution-based approach [9, 4]. MUP is the only one guaranteeing minimality of the core, whereas the only algorithm that scales well for large formal verification benchmarks is the resolution-based approach. We refer to the latter 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 [6], implement a DPLL backtrack search enhanced by a failure-driven assertion loop. These solvers explore the variable-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 for the given formula is found or when the empty clause ($\Box$)—signifying unsatisfiability—is derived. In the latter case, SAT solvers are able to produce a *resolution refutation* in the form of a directed acyclic graph (dag) $\Pi(V, E)$, whose vertices V are associated with clauses, and whose edges describe resolution relations between clauses. The vertices $V = V^i \cup V^c$ are composed of a subset V^i of the initial clauses and a subset V^c of the conflict clauses, including the empty clause $\Box$. The empty clause is the sink of the refutation graph, and the sources are V^i. Here, we understand a refutation to contain those clauses connected to $\Box$. The sources of the refutation comprise the unsatisfiable core returned by EC. Invoking EC until a fixed point is reached [9], allows one to reduce the unsatisfiable core even more. We refer to this algorithm as *EC-fp*. However, the resulting cores are still not guaranteed to be minimal and can be further reduced.

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. For every initial clause C in Π, check whether it belongs to a MUC in the following manner:
   (a) 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.
   (b) If they are satisfiable, then C belongs to a MUC, so continue with another initial clause.
   (c) 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.
3. Terminate when all the initial clauses remaining in Π comprise a MUC.

Our basic *Complete Resolution Refutation (CRR)* algorithm is described in Sect. 2, and a pruning technique, enhancing CRR and called *Resolution Refutation-based Pruning (RRP)*, is described in Sect. 3. Experimental results are presented and analyzed in Sect. 4. This is followed up by a brief conclusion.

2 The Complete Resolution Refutation (CRR) Algorithm

One says that a vertex D is *reachable* from vertex C in graph Π if there is a path (of 0 or more edges) from C to D. The sets of all vertices that are reachable and unreachable from C in Π are denoted $Re(\Pi, C)$ and $UnRe(\Pi, C)$, respectively. The *relative hardness* of a resolution refutation is the ratio between the total number of clauses and the number of initial clauses.

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.

```
1: Build a refutation $\Pi(V^i \cup V^c, E)$ using a SAT solver
2: while unmarked clauses exist in $V^i$ do
3:    $C \leftarrow PickUnmarkedClause(V^i)$
4:    Invoke a SAT solver on $G = UnRe(\Pi, C)$
5:    if $UnRe(\Pi, C)$ is satisfiable then
6:       Mark $C$ as a MUC member
7:    else
8:       Let $\Pi'(V^i_G \cup V^c_G, E_G)$ be the refutation built by the solver
9:       $V^i \leftarrow V^i \cap V^i_G$; $V^c \leftarrow (V^i_G \cup V^c_G) \setminus V^i$; $E \leftarrow E_G$
10: return $V^i$
```

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

By construction of Π, the $UnRe(\Pi, C)$ clauses were derived independently of C. To check whether C belongs to a minimal core, we provide the SAT solver with $UnRe(\Pi, C)$, including the conflict clauses. We are trying to *complete the resolution refutation* without using C as one of the sources. Observe that $\Box$ is always reachable from C; thus $\Box$ is never passed as an input to the SAT solver. We let the SAT solver try to derive $\Box$, using $UnRe(\Pi, C)$ as the input formula, or else prove that $UnRe(\Pi, C)$ is satisfiable.

In the latter case, we conclude that C must belong to a minimal 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. Otherwise, the SAT solver returns a valid resolution refutation $\Pi'(V^i_G \cup V^c_G, E_G)$, where $G = UnRe(\Pi, C)$. We cannot use Π' as is, as the refutation for the subsequent iterations, since the sources of the refutation may only be initial clauses of F. The necessary adjustments to the refutation are shown on line 9.

3 Resolution-Refutation-Based Pruning

In this section, we propose an enhancement of Algorithm CRR by developing resolution refutation-based pruning techniques for when the SAT solver is invoked on $UnRe(\Pi, C)$ to check whether it is possible to complete a refutation without C. We refer to the suggested technique as *Resolution Refutation-based*

Pruning (RRP). (We presume that the reader is familiar with the functionality of a modern SAT solver.)

An assignment σ *falsifies* a clause C if every literal of C is *false* under σ; it *falsifies* a set of clauses P if every clause $C \in P$ is falsified by σ. We claim that a model for $\mathit{UnRe}(\Pi, C)$ can only be found under a partial assignment that falsifies every clause in some path from C to the empty clause in $\mathit{Re}(\Pi, C)$. The reason is that otherwise there would exist a satisfiable vertex cut U in Π, contradicting the fact that the empty clause is derivable from U. (We omit a formal proof due to space limitations.)

Denote a subtree connecting C and $\Box$ by $\Pi{\restriction_C}$. The RRP technique is integrated within the decision engine of the SAT solver. The solver receives $\Pi{\restriction_C}$, together with the input formula $\mathit{UnRe}(\Pi, C)$. 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 coupled with 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 $\mathit{UnRe}(\Pi, C)$ exists. However, the SAT solver should continue its work to produce a refutation. (Refer to the full version of this paper for details.)

4 Experimental Results

We have implemented CRR and RRP in the framework of the VE solver. VE, a simplified version of the industrial solver Eureka, is similar to Chaff [3]. 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*) [8]. 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 the other experiments.

Table 1 summarizes the results of a comparison of the performance of two algorithms for suboptimal unsatisfiable core extraction and five algorithms for minimal unsatisfiable core extraction in terms of execution time and core sizes.

First, we compare algorithms for minimal unsatisfiable core extraction, namely, Naïve, MUP, plain CRR, and CRR enhanced by RRP. In preliminary experiments, we found that invoking suboptimal algorithms for trimming down the sizes of the formulas prior to MUC algorithm invocation is always useful. We used Naïve, combined with EC-fp and AMUSE, and MUP, combined with

Table 1. Comparing algorithms for unsatisfiable core extraction. Columns **Instance**, **Var** and **Cls** contain instance name, number of variables, and clauses, respectively. The next seven columns contain execution times (in seconds) and core sizes (in number of clauses) for each algorithm. The cut-off time was 24 hours (86,400 sec.). Column **Rel. Hard.** contains the relative hardness of the final resolution refutation, produced by CRR+RRP. Bold times are the best among algorithms guaranteeing minimality.

			Subopt.		**CRR**		**Naïve**		**MUP**	**Rel.**
Instance	**Var**	**Cls**	**EC**	**EC-fp**	**RRP**	**plain**	**EC-fp**	**AMUSE**	**EC-fp**	**Hard.**
4pipe	4237		9	171	**3527**	4933	24111	time-out	time-out	1.4
		80213	23305	17724	17184	17180	17182			
4pipe_1_ooo	4647		10	332	**4414**	10944	25074	time-out	mem-out	1.7
		74554	24703	14932	12553	12515	12374			
4pipe_2_ooo	4941		13	347	**5190**	12284	49609	time-out	mem-out	1.7
		82207	25741	17976	14259	14192	14017			
4pipe_3_ooo	5233		14	336	**6159**	15867	41199	time-out	mem-out	1.6
		89473	30375	20034	16494	16432	16419			
4pipe_4_ooo	5525		16	341	**6369**	16317	47394	time-out	mem-out	1.6
		96480	31321	21263	17712	17468	17830			
3pipe_k	2391		2	20	**411**	493	2147	12544	mem-out	1.5
		27405	10037	6953	6788	6786	6784	6790		
4pipe_k	5095		8	121	**3112**	3651	15112	time-out	time-out	1.5
		79489	24501	17149	17052	17078	17077			
5pipe_k	9330		16	169	**13836**	17910	83402	time-out	mem-out	1.4
		189109	47066	36571	36270	36296	36370			
barrel5	1407		2	19	93	**86**	406	326	mem-out	1.8
		5383	3389	3014	2653	2653	2653	2653		
barrel6	2306		35	322	**351**	423	4099	4173	mem-out	1.8
		8931	6151	5033	4437	4437	4437	4437		
barrel7	3523		124	1154	**970**	1155	6213	24875	mem-out	1.9
		13765	9252	7135	6879	6877	6877	6877		
barrel8	5106		384	9660	**2509**	2859	time-out	time-out	mem-out	1.8
		20083	14416	11249	10076	10075				
longmult4	1966		0	0	8	**7**	109	152	13	2.6
		6069	1247	1246	972	972	972	976	972	
longmult5	2397		0	1	74	**31**	196	463	35	3.6
		7431	1847	1713	1518	1518	1518	1528	1518	
longmult6	2848		2	13	**288**	311	749	2911	5084	5.6
		8853	2639	2579	2187	2187	2187	2191	2187	
longmult7	3319		17	91	6217	**3076**	6154	32791	68016	14.2
		10335	3723	3429	2979	2979	2979	2993	2979	

EC-fp. CRR performs best when combined with EC, rather than EC-fp. The sizes of the cores do not vary much between MUC algorithms, so we concentrate on a performance comparison. One can see that the combination of EC-fp and Naïve outperforms the combination of AMUSE and Naïve, as well as MUP. Plain CRR outperforms Naïve on every benchmark, whereas CRR+RRP outperforms Naïve on 15 out of 16 benchmarks (the exception being the hardest instance of *longmult*). This demonstrates that our algorithms are justified practically. Usually, the speed-up of these algorithms over Naïve varies between 4 and 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*). RRP improves performance on most instances. 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*, a family that is hard for CRR, and even harder for RRP due to the hardness of the resolution proofs of its instances.

Comparing CRR+RRP on one side and EC and EC-fp on the other, we find that CRR+RRP always produce smaller cores than both EC and EC-fp. The average gain on all instances of cores produced by CRR+RRP over cores produced by EC and EC-fp is 53% and 11%, respectively. The biggest average gain of CRR+RRP over EC-fp is achieved on Families *fvp-unsat.2.0* and *longmult* (18% and 17%, respectively). Unsurprisingly, both EC and EC-fp are usually much faster than CRR+RRP. However, on the three hardest instances of the barrel family, CRR+RRP outperforms EC-fp in terms of execution time.

5 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, if it belongs to a minimal unsatisfiable core. The algorithm reuses conflict clauses and resolution relations throughout its execution. We have demonstrated that the proposed algorithm is faster than currently existing ones for minimal unsatisfiable cores extraction by a factor of 6 or more on large problems with non-overly hard resolution proofs, and that it finds smaller unsatisfiable cores than suboptimal algorithms.

References

1. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. Fifth Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, pages 193–207, 1999.
2. R. Bruni. Approximating minimal unsatisfiable subformulae by means of adaptive core search. *Discrete Applied Mathematics*, 130(2):85–100, 2003.
3. Z. Fu, Y. Mahajan, and S. Malik. ZChaff2004: An efficient SAT solver. In *Proc. Seventh Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'04)*, pages 360–375, 2004.
4. E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, pages 10886–10891, 2003.
5. J. Huang. MUP: A minimal unsatisfiability prover. In *Proc. Tenth Asia and South Pacific Design Automation Conference (ASP-DAC'05)*, pages 432–437, 2005.
6. J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
7. Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. AMUSE: A minimally-unsatisfiable subformula extractor. In *Proc. 41st Design Automation Conference (DAC'04)*, pages 518–523, 2004.
8. M. N. Velev and R. E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. In *Proc. 38th Design Automation Conference (DAC'01)*, pages 226–231, 2001.
9. L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In *Prelim. Proc. Sixth Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'03)*, 2003.