MUP: A Minimal Unsatisfiability Prover

Jinbo Huang
Computer Science Department
University of California, Los Angeles
jinbo@cs.ucla.edu

Abstract—After establishing the unsatisfiability of a SAT instance encoding a typical design task, there is a practical need to identify its minimal unsatisfiable subsets, which pinpoint the reasons for the infeasibility of the design. Due to the potentially expensive computation, existing tools for the extraction of unsatisfiable subformulas do not guarantee the minimality of the results. This paper describes a practical algorithm that decides the minimal unsatisfiability of any CNF formula through BDD manipulation. This algorithm has a worse-case complexity that is exponential only in the treewidth of the CNF formula. We provide an empirical evaluation of the algorithm, highlighting its efficiency on a set of hard problems as well as its ability to work with existing subformula extraction tools to achieve optimal results.

I. Introduction

In a typical design task formulated as an instance of Propositional Satisfiability (SAT), a satisfying assignment for the Boolean formula, in Conjunctive Normal Form (CNF) by custom, represents a design solution. When the CNF is found unsatisfiable, on the other hand, a need arises to identify the causes of its unsatisfiability in order that a feasible design may be obtainable by revising its specifications. Practical tools take a step in this direction by extracting small unsatisfiable subformulas from the target CNF [6, 5, 29, 19].

An interesting example is given in [19] where an FPGA routing problem is formulated as SAT and found to have no solution. While unsatisfiability alone offers no clue as to its causes, on closer examination the CNF in question is discovered to have four *minimal unsatisfiable* (MU) subsets. Each of the first two encodes a local requirement that three particular nets are to be routed through a 2-track channel—a pigeonhole problem bound to be insoluble. The other MU subsets, by contrast, pinpoint a culprit net which apparently contributes to the unroutability of both channels. While such MU subformulas no doubt impart valuable information on possible ways to rectify the design, their being minimal, in particular, helps ensure that the conveyed information is not needlessly obscured.

Existing tools for extracting unsatisfiable subformulas, unfortunately, stop short of guaranteeing their minimality, and do indeed produce suboptimal results in this respect [6, 5, 29, 19]. This is understandable, to be sure, because checking minimal unsatisfiability is a hard problem known to be D^P-complete[1] [20].

Formally, a CNF formula is minimal unsatisfiable if it is unsatisfiable but becomes satisfiable with any clause removed. There has been extensive work toward efficient algorithms for checking minimal unsatisfiability, centering on the notion of *deficiency* of the CNF formula, which is the number of its clauses minus the number of its variables. Let MU(k) be all MU formulas with deficiency k. It was shown in [1] that for $k \leq 0$, MU(k) is the empty set—all MU formulas have positive deficiency. Algorithms were provided for MU(1) and MU(2) with quadratic and cubic complexity in [11, 8], respectively, and for general MU(k) with complexity $n^{O(k)}$ in [17, 13]. Finally, MU(k) was shown to be decidable in $O(2^k n^4)$ time, and thus *fixed-parameter tractable* with respect to k [25].

Although efficiency is guaranteed for small k, an algorithm of complexity $O(2^k n^4)$ may not be viable for problems arising from practical applications, where k is often unbounded. The pigeonhole problem we have mentioned, for example, has $k = O(h^3)$ in its standard CNF encoding, where h is the number of holes into which $h+1$ pigeons are to be placed. As k grows in a real-world situation, these problems will likely be better solved by a new algorithm that is susceptible to a more tractable parameter.

Such an algorithm is the main subject of this paper. Our proposed method combines the strengths of both theoretical work on *treewidth*-based computation [12] and practical advances in efficiently manipulating Binary Decision Diagrams (BDDs) [7, 24]. As a result, not only are we able to offer a formal guarantee on the complexity of the algorithm, but we show that our implementation of this algorithm, which we shall refer to as MUP (Minimal Unsatisfiability Prover), is able to quickly prove the minimal unsatisfiability of some particularly hard classical problems, of which even to prove the unsatisfiability alone can be difficult for state-of-the-art SAT solvers.

Furthermore, we point out that with a straightforward extension, this program also doubles as an MU subformula extractor. This means that MUP can be used as an

[1]The complexity class D^P can be defined as all languages that are the intersection of a language in NP and one in coNP [21, 20]. D^P-complete problems are both NP-hard and coNP-hard.

optimizer for any existing tool that extracts unsatisfiable subformulas. This feature is particularly useful for certain problems where running MUP on the original CNF may not be the most efficient choice. One example of this situation will be given in Section IV, where the CNF formulas have a relatively large number of clauses not contributing to the MU cores, and existing tools based on search and resolution, such as zCore [29] and AMUSE [19], can do a good job in trimming them down quickly.

The rest of the paper is organized as follows. We briefly review previous work on unsatisfiable subformula extraction in Section II, and describe in detail our minimal unsatisfiability prover in Section III. Section IV empirically evaluates the efficiency of the program as well as its ability to detect and optimize suboptimal results returned by existing tools. Section V concludes our presentation.

II. Previous Work

We briefly review three most recent algorithms that have been proposed for the extraction of unsatisfiable subformulas. The algorithm of Bruni and Sassano [6, 5] is based on a SAT search tree where branching occurs on clauses instead of variables. When a clause is selected for branching, one of its variables is chosen and instantiated so as to satisfy the clause. On backtracking from a conflict, the previously chosen variable is set to the opposite value and a new variable instantiated such that the clause is again satisfied. During this search a *hardness* measure is calculated for each clause based on how many times the clause has been visited, how many times a conflict has occurred on the clause, and the length of the clause. The "adaptive core search" then starts with an empty set and iteratively adds the *hardest* clauses to it until the set is unsatisfiable, or throws clauses out if the satisfiability of the set has not been determined after a certain number of branchings. As noted in [19], the quality of the cores extracted by this algorithm greatly depends on the settings of a few control parameters, which have to be hand-picked based on the specific problem.

An extension of the zChaff SAT solver from Princeton [18], the zCore program [29] utilizes the unsatisfiability proof produced by zChaff in the form of a resolution DAG. The roots of this DAG are original clauses of the CNF and every other node is the resolvent of its parents. A sink representing the empty clause then proves the unsatisfiability of the CNF. The set of roots having a path to this sink is hence an unsatisfiable subset of the CNF, which is identified and returned by zCore. This subformula can be passed back to zChaff and the whole process repeated until a fixed point is reached after some iterations.

Similar to zCore, the AMUSE program capitalizes on the resolution procedure involved during SAT search [19]. However, a new technique is used where each clause of the CNF is disjoined with a new variable. These variables serve as *selectors* for the clauses and together with the learning process of the SAT solver allow AMUSE to implicitly search for unsatisfiable subformulas. AMUSE was noted for its ability to locate different unsatisfiable cores upon repeated runs, as well as its inability to perform well on large formulas since a new variable is added for each clause, growing the search space considerably [19].

As we have mentioned, these existing programs do not guarantee the minimality of the formulas extracted. We present next a practical prover for minimal unsatisfiability, which can then double as an extractor of unsatisfiable subformulas that *are* guaranteed to be minimal.

III. A Minimal Unsatisfiability Prover

We will present the algorithm used by our prover in two steps. First, we propose a formal method that reduces minimal unsatisfiability of a CNF to model counting an augmented formula with auxiliary variables. Second, we show how this can be implemented as a variable elimination procedure on BDDs, which has a worst-case complexity exponential only in the *treewidth* of the original CNF. In the third part of this section we give a simple extension to the algorithm so that it can also be used to extract MU subformulas from an unsatisfiable CNF.

A. Augmenting the CNF

Let $\Delta = c_1 \wedge \ldots \wedge c_m$ be the CNF whose minimal unsatisfiability is in question, and $X = \{x_1, \ldots, x_n\}$ its variables. Let Δ_i be a subformula of Δ obtained by removing clause c_i. Our goal is to determine the unsatisfiability of Δ and the satisfiability of all Δ_i for $1 \leq i \leq m$—a total of $m+1$ theories to test. We start by introducing a set of new variables $Y = \{y_1, \ldots, y_k\}$, where we make $k = \lceil \log(m+1) \rceil$ for reasons that will soon be clear. Note that this also distinguishes our approach from that adopted by AMUSE where a total of m variables are added. The idea here is that we wish to construct a new formula Δ' over variables $X \cup Y$ such that each instantiation of Y will result in Δ' simplying to one of our $m+1$ target theories. This can be done by enlisting the minterms over variables Y.

Recall that a minterm over a set of k variables is a conjunction of k literals where each variable appears exactly once. When $k = 3$, for example, there are a total of 8 minterms: $y_1y_2y_3$, $y_1y_2\overline{y}_3$, $y_1\overline{y}_2y_3$, $y_1\overline{y}_2\overline{y}_3$, $\overline{y}_1y_2y_3$, $\overline{y}_1y_2\overline{y}_3$, $\overline{y}_1\overline{y}_2y_3$, $\overline{y}_1\overline{y}_2\overline{y}_3$ (the over-bar denotes negation; the conjunction symbols have been omitted).

Let M_Y be an array of all the 2^k minterms over variables Y in arbitrary order, and let M_Y^i denote its i^{th} element, the array index starting at 1. We will now construct the new formula Δ' by augmenting each clause c_i of CNF Δ with a disjunct which is a minterm over variables Y:

$$\Delta' = \bigwedge_{i=1}^{m} (M_Y^i \vee c_i).$$

Given any truth assignment α for variables Y, it is clear that there is a unique $1 \leq p(\alpha) \leq 2^k$ such that $M_Y^{p(\alpha)}$ evaluates to 1 and all other minterms $M_Y^i, i \neq p(\alpha)$, evaluate to 0. A minterm M_Y^i, when evaluating to 1 (0), has the effect of removing (retaining) the corresponding clause c_i with which it is disjoined in Δ'. Hence it is not hard to see that when Δ' simplifies under assignment α, we have

Lemma 1 $\Delta'|_\alpha = \begin{cases} \Delta_{p(\alpha)}, & 1 \leq p(\alpha) \leq m \\ \Delta, & \textit{otherwise.} \end{cases}$

In other words, out of the 2^k instantiations of variables Y, exactly m of them will lead to simplifications of Δ' which correspond to the m subformulas of CNF Δ obtained by removing a single clause; all other instantiations of Y will equate Δ' with Δ.

Using the existential quantification operator $\exists X$, we will now establish the following correspondence between the minimal unsatisfiability of CNF Δ and the number of models of $\exists X \Delta'$. Recall that $\exists X f(X, Y)$ is defined as the theory $f'(Y)$ whose models are exactly those of $f(X, Y)$ after references to the X variables are removed.

Theorem 1 *CNF Δ is minimal unsatisfiable iff $\exists X \Delta'$ has exactly m models over variables Y.*

Proof. Consider all the 2^k assignments α for variables Y. Assume that Δ is minimal unsatisfiable, which means that $\exists X \Delta = 0$ and $\exists X \Delta_i = 1$ for all $1 \leq i \leq m$. According to Lemma 1, therefore, we have

$$(\exists X \Delta')|_\alpha = \exists X(\Delta'|_\alpha) = \begin{cases} \exists X \Delta_{p(\alpha)} = 1, & 1 \leq p(\alpha) \leq m \\ \exists X \Delta = 0, & \text{otherwise.} \end{cases}$$

In other words, $\exists X \Delta'$ has exactly m models.

Now for the reverse direction assume that $\exists X \Delta'$ has exactly m models. This implies that Δ is unsatisfiable because otherwise we would have $\exists X \Delta = \exists X \Delta_i = 1$ for all $1 \leq i \leq m$ and $\exists X \Delta'$ would be a tautology with 2^k models. Furthermore, all Δ_i must be satisfiable because otherwise $(\exists X \Delta')|_\alpha = \exists X(\Delta'|_\alpha)$ would only be true for less than m assignments α according to Lemma 1, meaning that $\exists X \Delta'$ would have less than m models. ■

B. Variable Elimination with BDDs

To decide the minimal unsatisfiability of CNF Δ, it now remains to count the models of $\exists X \Delta'$. For this purpose we will first construct a BDD for $\exists X \Delta'$, which will be over variables Y and hence have a size of $O(m)$ because $|Y| = k = O(\log m)$. Counting the models of such a BDD can therefore be done in $O(m)$ time [7].

Recall that Δ' is a conjunction of augmented clauses, each in the form of $M_Y^i \vee c_i$, where M_Y^i is a minterm over variables Y and c_i is a disjunction of literals over variables X. Such a "clause" can be easily converted into a BDD, which has exactly $k+|c_i|+2$ nodes. The following augmented clause: $y_1 \overline{y}_2 y_3 \vee \overline{x}_1 \vee \overline{x}_2 \vee x_3$, for example, translates into the BDD depicted in Fig. 1.

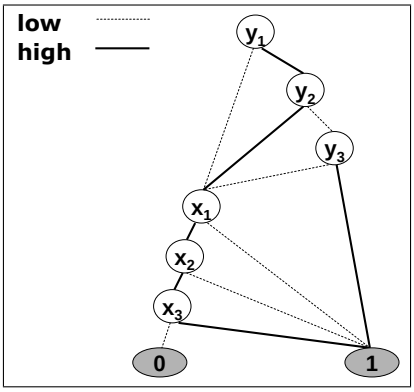

Fig. 1: A Binary Decision Diagram encoding $y_1 \overline{y}_2 y_3 \vee \overline{x}_1 \vee \overline{x}_2 \vee x_3$.

Algorithm 1 VE(conjuncts Δ', variables X, order π): constructs a BDD for $\exists X \Delta'$ by variable elimination

```
1: for each variable x ∈ X do
2:    create empty bucket B_x
3: for each conjunct c of Δ' do
4:    x = first X variable of c according to order π
5:    B_x = B_x ∪ {BDD(c)}
6: for each variable x ∈ X in order π do
7:    if B_x is not empty then
8:       bdd = conjunction of all elements of B_x in arbitrary order
9:       bdd = ∃x bdd
10:      if bdd = 0 then
11:         return 0
12:      else if bdd mentions a variable in X then
13:         x' = first X variable of bdd according to order π
14:         B_x' = B_x' ∪ {bdd}
15: return bdd
```

The next step is then to conjoin these m BDDs while existentially quantifying out the X variables. This calls for a well-studied technique known as *early quantification* [27, 9, 14], which has been extensively used in such areas as symbolic model checking where current state variables are quantified in image computation [9]. The idea is that one should take care to perform the quantifications as early as possible, rather than all at the end, so that the intermediate BDDs will have fewer variables and a likely smaller size. The rules of quantified Boolean logic, however, allow an existential quantifier to be pushed inside a conjunct only if the variable being quantified does not appear elsewhere. Under this constraint, identifying an optimal *quantification schedule* where the intermediate BDDs are minimized is not a trivial task [14].

For our purposes, we enlist the variable elimination (VE) procedure, which has been long established in the field of probabilistic inference [12]. Although VE has not traditionally been used for BDD construction, we shall see that it can indeed be a very efficient choice, and that it allows us to offer structure-based guarantees on the complexity of the algorithm.

Algorithm 1 describes VE as tailored specifically to our BDD construction task. It takes as inputs a list of conjuncts Δ' to be conjoined, a set of variables X to be elim-

inated (i.e., existentially quantified), and an ordering π on variables X, known as an *elimination order*. We start by converting each conjunct of Δ' into a BDD (implicit on Line 5), as described earlier and illustrated in Fig. 1.

We now create an empty *bucket* B_x for each variable x to be eliminated (Lines 1 and 2), and throw the BDD for each conjunct of Δ' into the bucket of its first X variable (recall that Δ' mentions both X and Y variables) according to order π (Lines 3–5). We then process the buckets using order π, skipping empty ones. When a bucket B_x is processed, all BDDs in it are conjoined in arbitrary order (Line 8) using the standard *Apply* operation [7], and variable x is existentially quantified (Line 9). The result is then thrown into the bucket of its first X variable (Lines 13 and 14), the exceptions being if a contradiction is encountered, which terminates the algorithm (Lines 10 and 11), or if the result only mentions Y variables, which is simply ignored (Line 12). The result of processing the last bucket will then be exactly a BDD for $\exists X \Delta'$ (Line 15).

We conclude this subsection by noting that VE algorithms are known to have a complexity that is exponential only in the *width* of the elimination order used, which corresponds to the *treewidth* of the CNF formula. We refer to reader to [12] for formal definitions of these notions, but point out here that in the context of Algorithm 1, the width of the elimination order π, plus 1, translates into the maximum number of variables any intermediate BDD can have. It is then easy to confirm that Algorithm 1 has time and space complextiy $O(m \cdot exp(w + \log m))$ where w is the width of order π with respect to the original CNF, because at most $m-1$ conjunctions need be performed and the *Apply* operation is linear in the product of its operand sizes [7], which are all bounded by $O(exp(w + \log m))$. Note that the extra $\log m$ introduced by the CNF augmentation only boils down to a polynomial factor. In Section IV we shall describe the method and tools we use to construct elimination orders of low width.

C. A Simple Extension

Now that we have a minimal unsatisfiability prover MUP, it is but natural to consider the possibility of using it also as a MU subformula extractor. When a CNF Δ is found by MUP to be unsatisfiable but not minimal, the BDD that has been constructed contains information about the satisfiability of all the m subformulas Δ_i. Specifically, each assignment α for the Y variables such that $1 \leq p(\alpha) \leq m$ and the BDD evaluates to 0 identifies subformula $\Delta_{p(\alpha)}$ as unsatisfiable. This information (one such assignment suffices) can be extracted in linear time, and the unnecessary clause $c_{p(\alpha)}$ removed. We can then repeatedly run MUP to shrink the resulting formula until it is proven to be minimal unsatisfiable. We point out that this method can be more efficient than it may appear, because the multiple runs of MUP are not completely independent—some work is shared through the caches maintained during BDD manipulation.

Before presenting our experimental results, we point out that running Algorithm 1 on the original CNF (i.e., without augmentation) will result in a BDD constant, because all variables will have been gone by the end of the elimination process. It is easy to see that this final constant will be 1 if and only if the CNF is satisfiable. One can therefore use this algorithm effectively as a SAT solver. The performance of such a solver has been favorably evaluated in [16] for many benchmarks, including some of those used here. Given this SAT solver, one can simply run it $m+1$ times, on the original CNF Δ and each of its m subsets Δ_i, as an alternative method for deciding minimal unsatisfiability. Note that, again, the multiple SAT tests can be more efficient than they appear owing to the shared caches. We will refer to this method as NAIVE and use it in the next section as a reference point to evaluate the efficiency of MUP.

IV. Experimental Results

Our first set of experiments involves running MUP and NAIVE on two classical families of hard SAT benchmarks—pigeonhole and Urquhart[2]—plus the whole *bevan* family (excluding the Urquhart instances) from SAT 2003 competition benchmarks [15], all of which are MU by construction. We use the CUDD package from the University of Colorado [24] for all BDD operations. To generate elimination orders of low width, which are essential for the efficiency of VE, we use the same method as described in [16]. This methond combines the benefits of two generation tools for low-width orders, HGR2BDT [10] and MINCE[3] [2], by running them both and choosing the result of smaller width.[4] All our experiments are run on a 2.4GHz processor with 3.7GB of RAM.

The results of these experiments are given in Table I, where all running times are in seconds, represent the group total, and include elimination order generation times. The number of instances in each group is given in the second column. For groups of multiple instances, the number of variables and the number of clauses are shown as a range.

It should be noted that these benchmarks have a history of baffling even the best SAT solvers [22]. The zChaff solver [18], from example, cannot determine the unsatisfiability of phole-13, phole-14, phole-15, ten of the *bevan* instances, or any of the urq-i instances for $i > 3$, within a 2000-second time limit on our computer.

[2]Each pigeonhole instance, phole-i, encodes the problem of placing $i+1$ pigeons in i holes such that no pigeons share a hole [15]. The Urquhart instances are also from [15] and based on the biconditional formulas described in [28].

[3]MINCE directly minimizes the *cutwidth*, not the width, but the two parameters are closely related through a third parameter known as *pathwidth*: $width \leq pathwidth \leq k \cdot cutwidth$ [4, 26], where k is the maximum clause length of the CNF minus 1.

[4]This combination seems to work well in general according to [16]. For the pigeonhole and automotive benchmarks (described later), however, we only use HGR2BDT as it seems to suffice.

TABLE II: Running MUP on Formulas Extracted by zCore

Benchmark			zCore Extraction				MUP Minimization		
Name	Variables	Clauses	Variables	Clauses	Time	Minimal?	Variables	Clauses	Time
C168_FW_SZ_107	1698	6599	42	50	0.09	N	41	47	0.03
C168_FW_UT_2468	1909	7487	32	36	0.07	N	32	35	0.03
C202_FS_RZ_44	1750	6199	12	19	0.06	N	12	18	0.01
C202_FS_SZ_84	1750	6273	206	221	0.09	Y	–	–	0.20
C202_FS_SZ_97	1750	6250	26	35	0.06	N	26	33	0.02
C202_FW_SZ_100	1799	8738	24	31	0.08	N	24	28	0.02
C202_FW_SZ_103	1799	10283	140	160	0.18	N	140	159	0.29
C202_FW_SZ_87	1799	8946	247	385	0.15	N	247	383	0.46
C202_FW_SZ_96	1799	8849	210	215	0.11	Y	–	–	0.19
C210_FS_SZ_55	1755	5781	29	49	0.06	N	29	46	0.02
C210_FW_SZ_90	1789	7994	221	284	0.14	Y	–	–	0.35
C210_FW_SZ_91	1789	7721	225	288	0.13	Y	–	–	0.35
C210_FW_UT_8630	2024	9721	23	38	0.11	N	23	35	0.02
C220_FV_SZ_55	1728	5753	246	312	0.19	N	246	310	0.43
C220_FV_SZ_65	1728	4496	24	30	0.04	N	24	29	0.01

TABLE I: Proving Minimal Unsatisfiability

Benchmark	Ins	Variables	Clauses	MUP	NAIVE
phole-6	1	42	133	0.23	0.26
phole-7	1	56	204	0.34	0.43
phole-8	1	72	297	0.51	0.99
phole-9	1	90	415	0.77	2.01
phole-10	1	110	561	1.07	3.60
phole-11	1	132	738	1.89	8.17
phole-12	1	156	949	2.10	16.32
phole-13	1	182	1197	4.81	28.32
phole-14	1	210	1485	13.58	109.33
phole-15	1	240	1816	83.86	10152.75
urq-3	10	36∼46	220∼470	2.54	8.36
urq-4	10	64∼87	356∼1030	5.84	40.25
urq-5	10	119∼127	978∼1336	24.18	137.51
urq-6	10	166∼180	1324∼1756	32.58	254.82
urq-7	10	229∼250	1880∼2420	61.95	596.07
urq-8	10	304∼327	2486∼3252	164.11	1391.95
dodecahedron	1	30	80	0.16	0.18
hcb2	1	12	32	0.09	0.09
hcb3	1	45	288	0.52	1.52
hcb4	1	112	2048	5.01	37.02
hcb5	1	225	12800	38.67	2472.37
hypercube4	1	32	128	0.21	0.24
hypercube5	1	80	512	1.13	2.29
hypercube6	1	192	2048	3.99	28.30
hypercube7	1	448	8192	22.20	526.12
icosahedron	1	30	192	0.22	0.32
icos_stretch	1	45	352	0.27	0.67
marg	17	12∼156	32∼1232	10.28	31.27

We observe that MUP, by contrast, is able to prove the unsatisfiability *and* minimality of all these CNFs, some instantaneously and the hardest one in just 84 seconds. NAIVE, on the other hand, is noticeably slower, although it still solves many instances in reasonable time. This appears to justify our CNF augmentation method, because all other techniques used are common to both programs.

In our second set of experiments, we run zCore to extract unsatisfiable subformulas from a suite of unsatisfiable benchmarks based on Automotive Product Configuration[5] [23, 3]. These CNFs "encode different consistency properties of the configuration data base which is used to configure DaimlerChrysler's Mercedes car lines [23, 3]." For each instance zCore is repeatedly called (via the provided run_till_fix script) until a fixed point is reached, that is, the unsatisfiable subformula cannot be minimized further. We then enlist MUP to check the minimality of the results and reduce those that are found nonminimal. Our goal here is to demonstrate the practicality and usefulness of MUP on these real-world problems, both in proving and in achieving optimality of the results.

The outcome of these experiments is summarized in Table II. There are a total of 84 benchmarks in the group. Due to space constraints we only include those instances where the subformula extracted by zCore has over 200 variables or is nonminimal. It can be seen that at an almost negligible cost, MUP is always able to either prove the minimality of the result, or reduce it so that it becomes minimal.

We note that running MUP directly on these benchmarks appears to be inefficient, apparently because there are a relatively large number of clauses not contributing to the unsatisfiable cores, which nonetheless must participate, in vain, in the BDD construction run by MUP. On the other hand, zCore as well as the other extractor AMUSE [19], both based on search and resolution, seem to do a good job in quickly trimming down the formulas.

[5]These benchmarks have also been used to evaluate AMUSE [19], which, however, is not available to us for empirical studies.

V. Conclusion

We have presented MUP—a minimal unsatisfiabilty prover. This program is based on a novel CNF augmentation method which reduces minimal unsatisfiability to a model counting problem. Capitalizing on theoretical and practical advances in such fields as treewidth-based computation, fast generation of high quality elimination orders, and engineering of efficient BDD packages, MUP is able to quickly prove the minimal unsatisfiability of a set of hard problems, of which even to prove unsatisfiability alone can be difficult for state-of-the-art SAT solvers. We have discussed the use of MUP as a minimal unsatisfiable core extractor. In situations where MUP by itself may not work best, we have demonstrated its ability to work with existing tools toward optimization of the results.

Acknowledgments

Thanks to the anonymous reviewers for their comments which will be valuable in extending our current results. This work has been partially supported by NSF grant IIS-9988543 and MURI grant N00014-00-1-0617.

References

[1] Ron Aharoni and Nathan Linial. Minimal non-two-colorable hypergraphs and minimal unsatisfiable formulas. *Journal of Combinatorial Theory Series A*, 43:196–204, 1986.

[2] Fadi Aloul, Igor Markov, and Karem Sakallah. Faster SAT and smaller BDDs via common function structure. In *International Conference on Computer Aided Design (ICCAD), University of Michigan*, 2001. Tool available for download at http://www.eecs.umich.edu/~faloul/Tools/mince/.

[3] SAT Benchmarks from Automotive Product Configuration, http://www-sr.informatik.uni-tuebingen.de/~sinz/DC/.

[4] H. L. Bodlaender, J. R. Gilbert, H. Hafsteinsson, and T. Kloks. Approximating treewidth, pathwidth, and minimum elimination tree height. *Journal of Algorithms*, 18:238–255, 1995.

[5] Renato Bruni. Approximating minimal unsatisfiable subformulae by means of adaptive core search. *Discrete Applied Mathematics*, 130(2):85–100, 2003.

[6] Renato Bruni and Antonio Sassano. Restoring satisfiability or maintaining unsatisfiability by finding *small* unsatisfiable subformulae. *Electronic Notes in Discrete Mathematics*, 9, 2001.

[7] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.

[8] Hans Kleine Büning. On subclasses of minimal unsatisfiable formulas. *Discrete Applied Mathematics*, 107(1–3):83–98, 2000.

[9] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *Proceedings of the International Conference on Very Large Scale Integration*, 1991.

[10] Adnan Darwiche and Mark Hopkins. Using recursive decomposition to construct elimination orders, jointrees and dtrees. In *Trends in Artificial Intelligence, Lecture notes in AI, 2143*, pages 180–191. Springer-Verlag, 2001.

[11] Gennady Davydov, Inna Davydova, and Hans Kleine Büning. An efficient algorithm for the minimal unsatisfiability problem for a subclass of CNF. *Annals of Mathematics and Artificial Intelligence*, 23(3–4):229–245, 1998.

[12] Rina Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence*, pages 211–219, 1996.

[13] Herbert Fleischner, Oliver Kullmann, and Stefan Szeider. Polynomial-time recognition of minimal unsatisfiable formulas with fixed clause-variable difference. *Theoretical Computer Science*, 289(1):503–516, 2002.

[14] R. Hojati, S. C. Krishnan, and R. K. Brayton. Early quantification and partitioned transition relations. In *Proceedings of the International Conference on Computer Design*, pages 12–19, 1996.

[15] Holger H. Hoos and Thomas Sttzle. SATLIB: An Online Resource for Research on SAT. In *I.P.Gent, H.v.Maaren, T.Walsh, editors, SAT 2000*, pages 283–292. IOS Press, 2000. SATLIB is available online at *www.satlib.org*.

[16] Jinbo Huang and Adnan Darwiche. Toward good elimination orders for symbolic SAT solving. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, November 2004.

[17] Oliver Kullmann. An application of matroid theory to the SAT problem. In *Proceedings of the 15th Annual IEEE Conference on Computational Complexity*, pages 116–124, 2000.

[18] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 39th Design Automation Conference*, 2001.

[19] Yoonna Oh, Maher N. Mneimneh, Zaher S. Andraus, Karem A. Sakallah, and Igor L. Markov. AMUSE: a minimally-unsatisfiable subformula extractor. In *Proceedings of the 41st Annual Conference on Design Automation*, pages 518–523. ACM Press, 2004.

[20] Christos H. Papadimitriou and David Wolfe. The complexity of facets resolved. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, pages 74–78, 1985.

[21] Christos H. Papadimitriou and Mihalis Yannakakis. The complexity of facets (and some facets of complexity). In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pages 255–260. ACM Press, 1982.

[22] The Annual SAT Competitions: http://www.satlive.org/SATCompetition/.

[23] Carsten Sinz, Andreas Kaiser, , and Wolfgang Kchlin. Formal methods for the validation of automotive product configuration data. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 17(2), 2003.

[24] Fabio Somenzi. CUDD: CU Decision Diagram Package. Release 2.4.0.

[25] Stefan Szeider. Minimal unsatisfiable formulas with bounded clause-variable difference are fixed-parameter tractable. In *Proceedings of the Nineth International Computing and Combinatorics Conference*, pages 548–558. Springer Verlag, 2003.

[26] Dimitrios Thilikos, Maria Serna, and Hans Bodlaender. A polynomial algorithm for the cutwidth of bounded degree graphs with small treewidth. *Lecture Notes in Computer Science*, 2161:380–390, 2001.

[27] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit enumeration of finite state machines using BDDs. In *Proceedings of the IEEE International Conferece on Computer Aided Design*, pages 130–133, 1990.

[28] Alasdair Urquhart. Hard examples for resolution. *Journal of the ACM*, 34(1):209–219, 1987.

[29] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable Boolean formula. Presented at the Sixth International Conference on Theory and Applications of Satisfiability Testing, 2003.