AMUSE: A Minimally-Unsatisfiable Subformula Extractor

Yoonna Oh, Maher N. Mneimneh, Zaher S. Andraus, Karem A. Sakallah, Igor L. Markov
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122

{yoh, maherm, zandrawi, karem, imarkov}@eecs.umich.edu

ABSTRACT

This paper describes a new algorithm for extracting unsatisfiable subformulas from a given unsatisfiable CNF formula. Such unsatisfiable "cores" can be very helpful in diagnosing the causes of infeasibility in large systems. Our algorithm is unique in that it adapts the "learning process" of a modern SAT solver to identify unsatisfiable subformulas rather than search for satisfying assignments. Compared to existing approaches, this method can be viewed as a bottom-up core extraction procedure which can be very competitive when the core sizes are much smaller than the original formula size. Repeated runs of the algorithm with different branching orders yield different cores. We present experimental results on a suite of large automotive benchmarks showing the performance of the algorithm and highlighting its ability to locate not just one but several cores.

Categories and Subject Descriptors

G.2.1 [Combinatorics]: Combinatorial Algorithms.

General Terms

Algorithms, Verification.

Keywords

Minimally-unsatisfiable subformula, (MUS), conjunctive normal form (CNF), Boolean satisfiability (SAT), diagnosis.

1 INTRODUCTION

Formulations of electronic design automation tasks as instances of Boolean satisfiability (SAT) fall into two categories. In *verification* applications, a large Boolean function is formed such that it is satisfiable when the object being verified contains bugs. SAT solving, then, reveals the existence of bugs when the function is satisfiable or establishes their absence when the function is unsatisfiable. An example of this is the functional verification of hardware (equivalence or property checking) [8]. In *design* applications, a large Boolean function is formed such that a feasible design is obtained when the function is satisfiable, and design infeasibility is indicated when the function is unsatisfiable. An example of this is the routing of signal wires in an FPGA [9]. Unlike the verification scenario where unsatisfiability establishes a proof of correctness, unsatisfiability in the design context implies a negative result; without further analysis of the causes of unsatisfiability, we have no clue as to how to relax the design constraints in order to obtain a feasible design. The goal of this paper is to analyze the causes of unsatisfiability in large CNF formulas in order to provide useful diagnostic information to designers pinpointing those design constraints that must be modified.

DAC'04, June 7-11, 2004, San Diego, California, USA

Consider an unsatisfiable CNF formula φ. An *unsatisfiable subformula* (a US) of φ is a *minimally-unsatisfiable subformula* (an MUS) if it becomes satisfiable whenever any of its clauses is removed. An unsatisfiable CNF formula can have one or more MUSes, and the set of all MUSes is referred to as the formula's *clutter* [2].

Theoretical work on minimal unsatisfiability. Papadimitriou and Wolfe [10] showed minimal unsatisfiability to be DP-complete, where DP is the class which can be described as the difference between two NP problems. A DP-complete problem is equivalent to solving a SAT-UNSAT problem defined as: given two formulas φ and ψ, in CNF, is it the case that φ is satisfiable and ψ is unsatisfiable? Minimally unsatisfiable formulas always have positive *deficiency* [1, 3], where deficiency is the difference between the number of clauses and variables. Davydov et al. [3] gave an efficient algorithm for recognizing minimally-unsatisfiable formulas with deficiency 1. Kleine Buning [5] showed that, if k is a fixed integer, then the recognition problem with deficiency k is in NP, and suggested a polynomial time algorithm for formulas with deficiency 2. Recently, it was shown that minimal unsatisfiable formulas with deficiency k can be recognized in time $n^{O(k)}$ where n is the number of variables [4].

Experimental work on minimal unsatisfiability. Bruni and Sassano [2] employ an "adaptive core search" procedure that ranks clauses based on their *hardness*. The hardness of a clause is defined as a weighted sum of how often the clause is visited during a complete search algorithm and how often it is involved in conflicts. Starting from a small initial set 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 (chosen based on hardness) until the core becomes unsatisfiable. The process uses three parameters: d, the number of branching steps used to obtain the initial core; b, the number of branching steps in subsequent iterations; and c, the percentage of clauses used to expand or contract the current core. Not surprisingly, the quality (i.e., size and minimality) of the unsatisfiable core produced by this procedure is highly dependent on the particular settings of these three parameters.

Zhang and Malik [15] generate a resolution proof from the SAT solver and use it to derive an unsatisfiable subformula. A resolution proof is a DAG whose source vertices (i.e., vertices with no incoming edges) correspond to the formula's original clauses. Each other vertex has exactly two predecessors and corresponds to the consensus clause obtained from those of its predecessors. The set of original clauses in the transitive fan-in of the sink are returned as the unsatisfiable core. The method works on very large instances (by storing the resolution DAG on disk) but the returned core is not guaranteed to be an MUS. Smaller cores, though not necessarily MUSes, can be obtained by repeated application of the above procedure until no further reduction in size is obtained.

In this paper we propose a new approach to finding a set of unsatisfiable subformulas from a given unsatisfiable formula. The approach capitalizes on the conflict-driven learning of modern backtrack SAT solvers such as GRASP [7] and zChaff [14] to identify unsatisfiable subformulas in a bottom-up fashion during the search. The rest of the paper is organized as follows. In Section 2 we motivate the need to extract MUSes using a small example. The main extraction algorithm is described in Section 3. Empirical evaluation of the algorithm on a set of large industrial automotive benchmarks is given in Section 4, and the paper ends in Section 5 with conclusions and some directions for future work.

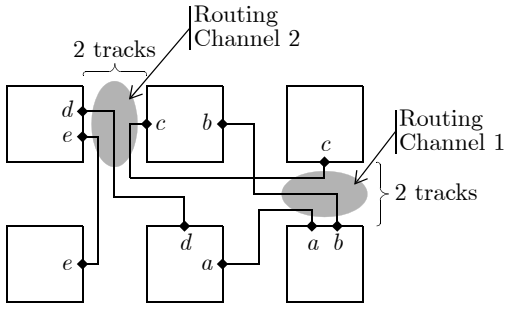
2 A MOTIVATING EXAMPLE

Most previous work on minimally-unsatisfiable formulas was concerned with either proving that an unsatisfiable formula is minimal, or extracting a minimally (preferably the smallest possible) unsatisfiable subformula from it. We argue here that it is useful to extract not just one but several MUSes from a given formula. Furthermore, we argue that larger MUSes may sometimes be more useful for diagnostic purposes than smaller ones.

Consider the small FPGA routing problem shown in Figure 1(a). As indicated, there are five nets to be routed (a through e) over an FPGA fabric that has two tracks (numbered 0 and 1) in each routing channel. A net x is modeled by two binary variables x_0 and x_1 that indicate its track assignment: $x_i = 1$ indicates that net x is assigned to track i. With this encoding, the routing requirements are now formulated as a set of CNF clauses that fall into one of two categories [9]:

1. *Liveness* constraints to insure that each net is assigned to at least one routing track. There are five such constraints, one per net.
2. *Exclusivity* constraints to insure that each track is assigned to at most one net. There are twelve such constraints, six for routing channel 1 and six for routing channel 2.

The resulting (unsatisfiable) set of seventeen constraints is indicated in Figure 1(b) along with four of its MUSes. The smaller of these MUSes, MUS_1 and MUS_2, are clearly seen to correspond to routing channels 1 and 2, respectively. As a means of diagnosing the causes of unroutability, MUS_1, for instance, pinpoints the conflicting requirements of trying to route three nets (a, b, and c) in a 2-track channel. Similarly, MUS_2 indicates the impossibility of routing nets c, d, and e in channel 2. It is interesting, and perhaps not surprising, that MUS_1 and MUS_2 are pigeon hole instances with 3 pigeons (nets) and 2 holes (tracks) each. In contrast, MUS_3 and MUS_4 are not pigeon hole instances. Rather than pinpointing a routing channel whose capacity is exceeded, these MUSes seem to pinpoint a culprit net, namely c, that contributes to the unroutability of both channels. This can be seen by noting that the c variables occur more frequently in these MUSes than do all other variables.



(a) A small FPGA routing problem. The square boxes represent configurable logic blocks and the spaces between them correspond to routing channels.

		MUS_1	MUS_2	MUS_3	MUS_4
Liveness Constraints	$C_1 : (a_0 + a_1)$	●		●	●
	$C_2 : (b_0 + b_1)$	●		●	●
	$C_3 : (c_0 + c_1)$	●	●	●	●
	$C_4 : (d_0 + d_1)$		●	●	●
	$C_5 : (e_0 + e_1)$		●	●	●
Channel 2 Exclusivity Constraints	$C_6 : (c'_0 + d'_0)$		●		●
	$C_7 : (c'_0 + e'_0)$		●		●
	$C_8 : (d'_0 + e'_0)$		●	●	
	$C_9 : (c'_1 + d'_1)$		●	●	
	$C_{10} : (c'_1 + e'_1)$		●	●	
	$C_{11} : (d'_1 + e'_1)$		●		●
Channel 1 Exclusivity Constraints	$C_{12} : (a'_0 + b'_0)$	●			●
	$C_{13} : (a'_0 + c'_0)$	●		●	
	$C_{14} : (b'_0 + c'_0)$	●		●	
	$C_{15} : (a'_1 + b'_1)$	●		●	
	$C_{16} : (a'_1 + c'_1)$	●			●
	$C_{17} : (b'_1 + c'_1)$	●			●

(b) Routing constraints and corresponding MUSes

Figure 1: An FPGA routing example and four of its MUSes

This simple example brings out several interesting points about MUSes and the role they might play in diagnosing and eliminating the causes of infeasibility. "Small" MUSes, such as MUS_1 and MUS_2, might be useful in identifying "local infeasibility." In this example, the local infeasibility identified by MUS_1 is in channel 1; it can be eliminated by increasing channel capacity or by re-routing one of the nets. Symmetry, however, prevents MUS_1 from providing any guidance as to which net to re-route (MUS_1 remains invariant under any permutation of the three nets.) Thus, if a net

other than c is chosen for re-routing, the problem would remain infeasible as the other channel is still congested[1]. On the other hand, "large" MUSes, such as MUS_3 and MUS_4, require deeper analysis to identify the cause of infeasibility. They are more global, though, and suggest corrective actions that are more likely to eliminate infeasibility more efficiently; in this case, re-routing net c solves the congestion in both channels simultaneously. It is also interesting to note that MUS_3 and MUS_4 are in some sense "equivalent" in that they convey the same information (namely, that net c is problematic.)

Clearly, further research is needed to understand the exact relation between a formula's MUSes and their effectiveness in pinpointing the reasons for infeasibility as well as the best corrective actions to eliminate it. Our concern in this paper is to devise efficient algorithms for extracting several, if not all, MUSes/USes from a given unsatisfiable formula.

3 THE AMUSE ALGORITHM

Consider an unsatisfiable formula φ. A systematic, albeit expensive, algorithm for finding φ's clutter is to build a search tree (the *MUS tree*) that enumerates φ's subformulas. Each node in this tree corresponds to a subformula of φ, with the root corresponding to the entire formula. A node in the MUS tree is *expanded* if its corresponding subformula is unsatisfiable. Expansion of a node whose associated subfomula is $\psi = C_1 \cdot C_2 \cdot \cdots \cdot C_l$, denotes the creation of l child nodes such that the i-th child corresponds to the subformula obtained by removing clause C_i from ψ. The subformula at a node is an MUS if the subformulas at all of the node's children are satisfiable.

Several enhancements to the above algorithm are possible. For instance, the SAT checks at each node can be performed incrementally [13] to allow the sharing of conflict clauses among subformulas. In addition, redundant SAT checks can be quickly identified and eliminated by caching techniques (effectively transforming the tree into a DAG.) The multiple satisfiability checks at the children of each node can also be reduced to a single satisfiability check of a larger formula. Finally, we may choose to expand the tree partially in order to identify various subsets of the clutter. Despite all of these enhancements, however, construction of the MUS tree is infeasible except for very small formulas.

The MUS tree algorithm can be viewed as a *top-down* solution to finding MUSes: an MUS is computed by monotonically shrinking (i.e., removing clauses from) the original formula. This algorithm exhibits its worst-case behavior when the size of a formula (i.e., the number of its clauses) is much larger than the sizes of its MUSes; such MUSes would not be found until the MUS tree has been expanded to very deep levels. In such cases, a *bottom-up* approach should perform better: starting from an empty formula, clauses are added until a US is obtained. While such a process does not, in general, guarantee the identification of an MUS, it can leverage the learning power of a modern SAT solver to increase the likelihood of including only relevant clauses in the evolving US. This is the key idea behind the AMUSE algorithm.

[1] It is interesting to note, though, that the intersection of MUS_1 and MUS_2 is the liveness clause for net c.

Algorithm 1 AMUSE: A Minimally Unsatisfiable Subformula Extractor

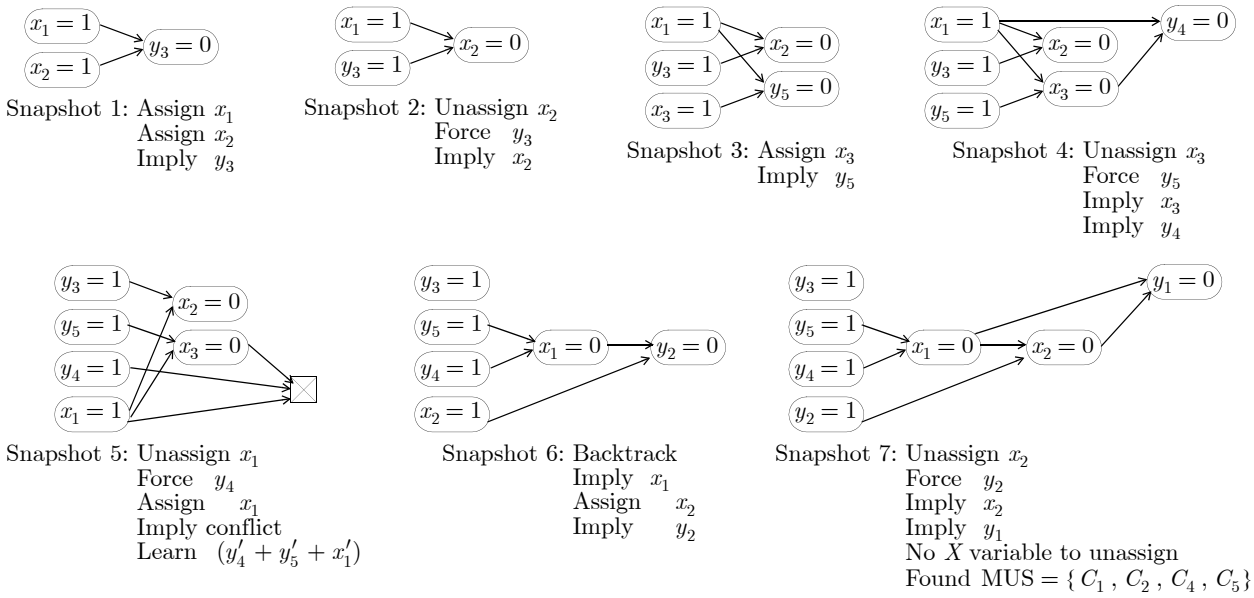
```
bool yWasImplied;
stack xWasUnassigned;
DLLSearch() {
    while (true)
        if (Decide() == false)
            return SATISFIABLE;
        while (Deduce() == CONFLICT)
            if (Diagnose() == CONFLICT)
                return UNSATISFIABLE;}
Decide() {
    if (no unassigned x∈X)
        return false;
    if (yWasImplied == true)
        Find implied y_i;
        assign y_i to 1;
        yWasImplied = false;
    else if (xWasUnassigned != empty)
        pop unassigned x_i;
        assign x_i;
    else
        choose unassigned x_i∈X;
        assign x_i;
    return true;}
Deduce() {
    result = boolean_constraint_propagation();
    if (yWasImplied)
        return CONFLICT;
    if (result == CONFLICT)
        return CONFLICT;
    return SUCCESS;}
Diagnose() {
    if (yWasImplied)
        find_x_to_unassign();
        if (no such x)
            Record MUS solution;
        else
            Unassign x;
            push x onto xWasUnassigned;
            return SUCCESS;
    else
        learn_conflict_clause();
        if (!backtrack_until_no_conflict())
            return CONFLICT;
        return SUCCESS;}
```

Figure 2: The AMUSE algorithm

The algorithm extends a generic DLL-based SAT solver to *implicitly* search for a US instead of a satisfying assignment. This is achieved by introducing a set of auxiliary variables, one per clause, that serve as clause *selectors*. Consider the unsatisfiable formula $\varphi(X) = C_1(X) \cdot C_2(X) \cdot \cdots \cdot C_k(X)$ where $X = \{x_1, x_2, \ldots, x_n\}$, and define the following associated *satisfiable* formula:

$$\begin{aligned}\hat{\varphi}(X,Y) &= (y_1 \rightarrow C_1(X)) \cdot (y_2 \rightarrow C_2(X)) \cdots (y_k \rightarrow C_k(X)) \\ &= (y_1' + C_1(X)) \cdot (y_2' + C_2(X)) \cdots (y_k' + C_k(X))\end{aligned}$$

where $Y = \{y_1, y_2, \cdots, y_k\}$ and $Y \cap X = \varnothing$. Note that by setting $y_i = 1$, clause C_i is *activated* or included, and by setting $y_i = 0$ clause C_i is *deactivated* or excluded. Intuitively, $\hat{\varphi}(X,Y)$ implicitly encodes the entire set of $\varphi(X)$'s subformulas. Thus, the problem of finding a US of $\varphi(X)$ reduces to finding a truth assignment Y^* such that $\hat{\varphi}(X,Y^*)$ is unsatisfiable.



$$\hat{\varphi}(X,Y) = (y'_1 + x_1 + x_2)(y'_2 + x_1 + x'_2)(y'_3 + x'_1 + x'_2)(y'_4 + x'_1 + x_3)(y'_5 + x'_1 + x'_3)$$

Figure 3: Execution snapshots of the AMUSE algorithm on a sample formula

The AMUSE algorithm operates on $\hat{\varphi}(X,Y)$. Figure 2 summarizes its pseudo code, with the text in bold indicating the extensions to a generic SAT solver. AMUSE distinguishes between the X and Y variables and treats them differently during the search. Specifically, the X variables are handled normally, i.e., they are assigned electively (by the decision process) or forcibly (by the deduction process), and unassigned during backtracking after conflicts. The Y variables, however, are viewed as "meta variables" that act to identify clauses that should be collected in order to generate a US. The algorithm *forces* a Y variable to assume the value 1 to indicate that the corresponding clause is a candidate for inclusion in the evolving US. To understand how this is accomplished, let X_{decided} and X_{implied} denote the set of decided and implied X variables and let Y_{forced} represent the set of forced Y variables. Also, let the union of X_{decided} and X_{implied} be denoted by X_{assigned}. Initially, these sets are empty. As the search progresses, the set X_{assigned} is extended. Since the formula is unsatisfiable, at some point we will have $C_i(X_{\text{assigned}}) = 0$ for some clause C_i forcing the deduction process (Boolean Constraint Propagation) to set $y_i = 0$ in $\hat{\varphi}(X,Y)$. This indicates that clause C_i must be deactivated for the SAT solver to find a satisfying truth assignment for $\hat{\varphi}(X,Y)$ and identifies it as a potential candidate for the evolving US. This is now done by unassigning a variable in X_{decided} that participated in the implication of y_i to 0 and forcing y_i to 1 instead. At this point, the search resumes normally until the combination of X_{assigned} and Y_{forced} cause a conflict and result in the creation of a conflict-induced clause ω. Because $\hat{\varphi}(X,Y)$ is satisfiable (trivially by setting all of the Y variables to 0), ω is guaranteed to have at least two negative Y literals. When the appropriate X decision variable is unassigned, this clause allows the search process to backtrack and to cause the implication of that unassigned variable to the opposite value.

As the search continues, we are guaranteed, by the unsatisfiability of $\varphi(X)$, to trigger the indirect implication of a Y variable, y_{ii}, to 0 exclusively by a set of other Y variables, $y_{i1}, y_{i2}, \cdots, y_{i,j-1}$. This implication can be recorded as the clause $(y'_{i1} + y'_{i2} + \cdots + y'_{ij})$ which implicitly identifies a US. Execution of the AMUSE algorithm on a sample formula is illustrated in Figure 3.

It is instructive to note that different decision heuristics on the X variables usually lead to the generation of different USes. To generate a user-specified number of USes, we re-run AMUSE by favoring variables that did not appear in previously-generated USes. This is achieved by storing priority information for each variable. After a US is found, the priority of variables appearing in this US is decremented, and the algorithm is run again. We can also optimize the algorithm to increase the likelihood that the generated US is in fact an MUS. One way of achieving this is for the decision process to favor variables in activated clauses with the least number of unassigned literals.

4 EXPERIMENTAL EVALUATION

We implemented the AMUSE algorithm on top of the MiniSAT solver [6] and tested it on a number of large unsatisfiable benchmarks for Automotive Product Configuration [11, 12]. These benchmarks model the configuration of options for the DaimlerChrysler Mercedes car and truck lines. The benchmark suite consists of 84 unsatisfiable instances that range in size from 1608 variables and 4496 clauses to 2038 variables and 11,352 clauses. We conducted our experiments on a 2.8 GHz Pentium IV PC with 1 GB of RAM running the Linux operating system. Table 1 lists the results of running AMUSE and zCore, the zChaff core extractor [15], on some of these benchmarks. For each benchmark, the table lists its name (col. 1), the number of its variables and clauses (cols. 2 and 3), and the results of running AMUSE and zCore. For each of these extractors, the table lists the num-

Table 1: Unsatisfiable Subformula Results for the DaimlerChrysler Product Configuration Benchmarks

Benchmark			AMUSE					zCore				
Name	#V	#C	#I	#V	#C	Time, sec.	MUS?	#I	#V	#C	Time, sec	MUS?
C168_FW_UT_851	1909	7491	1	7	8	0.01	Y	2	9	10	0.07	Y
C168_FW_UT_852	1909	7489	1	7	8	0	Y	2	9	10	0.07	Y
C168_FW_UT_854	1909	7486	1	7	8	0	Y	2	9	10	0.05	Y
C168_FW_UT_855	1909	7485	1	7	8	0.01	Y	2	9	10	0.06	Y
C208_FA_RZ_43	1608	5297	1	6	8	0	Y	1	8	9	0.04	Y
C202_FW_SZ_98	1799	8689	1	6	9	0.01	Y	1	6	8	0.06	Y
C168_FW_UT_714	1909	7487	1	6	9	0.01	Y	1	6	9	0.04	Y
C220_FV_RZ_13	1728	4509	1	9	10	0	Y	1	9	10	0.04	Y
C220_FV_RZ_12	1728	4512	1	10	11	0.01	Y	1	10	11	0.02	Y
C220_FV_RZ_14	1728	4508	1	10	11	0	Y	1	10	11	0.03	Y
C208_FC_RZ_65	1654	5591	1	11	12	0	Y	2	13	15	0.05	Y
C210_FS_SZ_107	1755	5762	2	11	15	0.01	Y	2	11	15	0.04	Y
C220_FV_SZ_46	1728	4498	1	16	17	0	Y	1	16	17	0.02	Y
C208_FA_SZ_87	1608	5299	1	17	18	0.01	Y	1	17	20	0.03	Y
C210_FW_SZ_111	1789	7404	1	13	18	0	Y	2	11	15	0.06	Y
C202_FS_RZ_44	1750	6199	2	14	20	0	Y	2	12	19	0.04	N
C202_FS_SZ_121	1750	6181	1	21	23	0	Y	1	20	22	0.04	Y
C210_FW_SZ_128	1789	7412	2	12	23	0	Y	2	14	23	0.06	Y
C220_FV_SZ_65	1728	4496	1	18	23	0.01	Y	1	24	30	0.03	N
C202_FS_SZ_95	1750	6184	2	20	26	0	Y	1	12	14	0.04	Y
C202_FS_SZ_104	1750	6201	2	23	28	0	Y	1	29	34	0.04	Y
C208_FA_RZ_64	1608	5279	1	29	212	0.01	Y	1	29	212	0.03	Y
C208_FC_RZ_70	1654	5543	1	29	212	0.01	Y	2	29	212	0.04	Y
C202_FW_RZ_57	1799	8685	1	30	213	0	Y	1	30	213	0.05	Y
C202_FW_SZ_96	1799	8849	1	218	223	0	Y	2	210	215	0.08	Y
C202_FS_SZ_84	1750	6273	1	207	226	0.01	Y	1	206	221	0.05	Y
C170_FR_RZ_32	1659	4956	1	30	227	0	Y	2	30	227	0.06	Y
C210_FW_SZ_129	1789	7606	2	62	234	0.01	Y	1	49	176	0.04	Y
C210_FW_SZ_90	1789	7994	2	228	294	0.05	N	1	221	284	0.09	Y
C210_FW_SZ_91	1789	7721	2	212	299	0.06	N	1	225	288	0.08	Y
C220_FV_SZ_114	1728	4777	2	66	323	0.03	N	1	47	132	0.03	Y
C220_FV_SZ_55	1728	5753	2	237	394	0.03	N	2	246	312	0.15	N
C202_FW_SZ_87	1799	8946	3	246	416	0.05	N	2	247	385	0.08	N

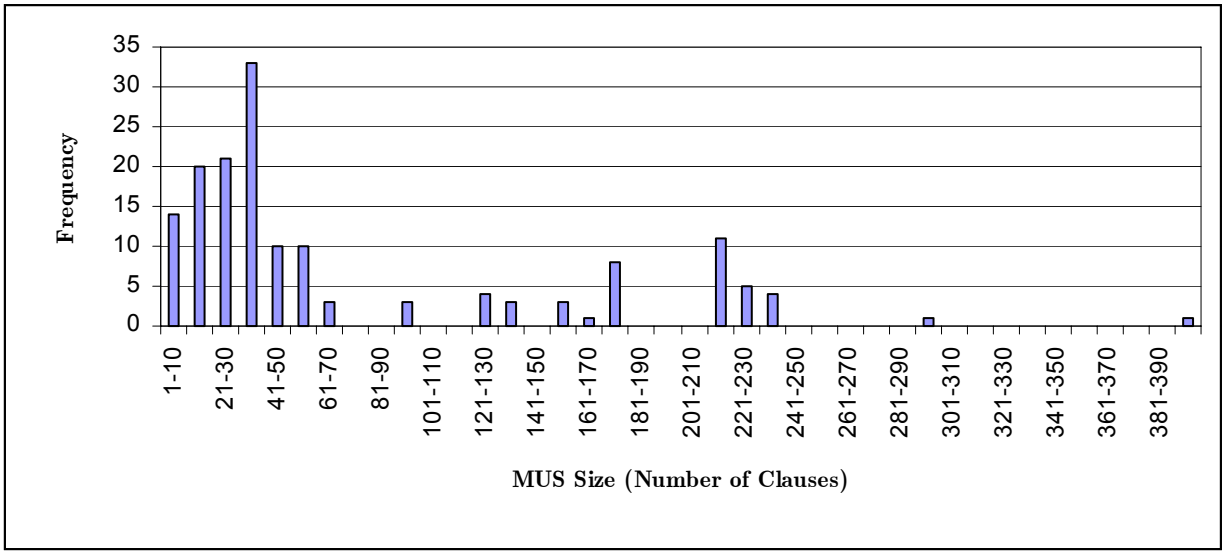


Figure 4: Distribution of MUS sizes for the DaimlerChrysler Product Configuration Benchmarks

ber of iterations needed to yield the smallest possible US, the size of the US (its number of variables and clauses), the extraction time, and whether the returned US is an MUS. The table is sorted by the size of the US returned by AMUSE, and we only show the first 30 and last 12 benchmarks due to space limitations. These data suggest that AMUSE is generally able to extract small MUSes in one or two iterations; for larger unsatisfiable cores AMUSE is less successful in extracting MUSes. In all cases, AMUSE seems to run several times faster than zCore.

To illustrate AMUSE's capability of identifying several USes/MUSes from a given formula, we employed the technique described at the end of Section 3 to generate up to four MUSes from each benchmark formula. The size distribution of these MUSes is shown in Figure 3. Except for a small cluster of MUSes with sizes between 200 and 220, most MUSes are less than 70 clauses in size. The most common MUS size is between 31 and 40 clauses (recall that these benchmarks have thousands of clauses.)

5 CONCLUSIONS AND FUTURE WORK

With the increasing use of SAT solvers in design automation flows, there is an emerging need for diagnosing and explaining the causes of failure. The AMUSE algorithm adapts the learning process of a modern SAT solver to identify unsatisfiable subformulas in a bottom-up fashion during the search. We anticipate active research in this area for the next few years as we strive to understand the "structure" of unsatisfiability and how it maps back to various application domains. Some of the issues that will be interesting to explore include:

- Developing a better understanding of the relation between MUSes, which are manifestations of infeasibility, and their relation to the causes of infeasibility. This will help pinpoint the minimal changes that need to be made in a system to restore feasibility and will involve studying possible covering relations among MUSes.
- Computationally, the AMUSE algorithm does not scale well for large formulas as it has to add a y variable to each clause. This can be addressed in a number of ways such as using a resolution-based core extractor [15] as a front end, or ranking the clauses based on their difficulty as in [2] and adding y variables to a certain number of difficult clauses. In addition, recent work on symmetry detection and breaking may be applicable as a way to a) speed up the search for MUSes, and b) reduce the number of extracted MUSes by collapsing "equivalent" MUSes (equivalence being informally related to their diagnostic abilities). Finally, structural decomposition of large formulas, using techniques from hypergraph partitioning, might be helpful in further scaling the applicability of AMUSE.
- Another interesting feature of the AMUSE algorithm is its ability to prove unsatisfiability. Checking a large instance for unsatisfiability might be infeasible for a SAT solver. However, if the instance has small MUSes, AMUSE can prove unsatisfiability if it can find such MUSes quickly.

ACKNOWLEDGMENTS

This work was funded in part by the DARPA/MARCO Gigascale Systems Research Center, and in part by the National Science Foundation under ITR grant No. 0205288. The authors would also like to acknowledge Fadi Aloul for his help in the early stages of this project.

REFERENCES

[1] R. Aharoni and N. Linial, "Minimal Non-Two-Colorable Hypergraphs and Minimal Unsatisfiable Formulas," in *J. Combinatorial Theory, Series A*, vol. 43, 1986.

[2] R. Bruni and A. Sassano, "Restoring Satisfiability or Maintaining Unsatisfiability by finding *small* Unsatisfiable Subformulae," in *Electronic Notes in Discrete Mathematics*, vol. 9, 2001.

[3] G. Davydov, I. Davydova, and H. K. Buning, "An Efficient Algorithm for the Minimal Unsatisfiability Problem for a Subclass of CNF," in *Annals of Mathematics and Artificial Intelligence*, vol. 23, pp. 229-245, 1998.

[4] H. Fleischner, O. Kullmann, and S. Szeider. "Polynomial-Time Recognition of Minimal Unsatisfiable Formulas with Fixed Clause-Variable Difference," in *Theoretical Computer Science*, 289(1), pp.503-516, 2002.

[5] H. Kleine Buning, "On Subclasses of Minimal Unsatisfiable Formulas," in *Discrete Applied Mathematics*, 197(1-3), pp. 83-98, 2000.

[6] N. Eén, and N. Sörensson, "An Extensible SAT-solver," in *Sixth International Conference on Theory and Applications of Satisfiability Testing* (SAT), 2003.

[7] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," in *IEEE Transactions on Computers*, vol. 48, 1999.

[8] K. L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*, Kluwer Academic Publishers, 1993.

[9] G. Nam, F. A. Aloul, K. A. Sakallah, and R. A. Rutenbar, "A Comparative Study of Two Boolean Formulations of FPGA Detailed Routing Constraints," in P*roceedings of the ISPD* 2001.

[10] C. H. Papadimitriou, and D. Wolfe, "The Complexity of Facets Resolved," in *J. Computer and System Sciences*, vol. 37, pp. 2-13, 1988.

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

[12] C. Sinz, A. Kaiser, and W. Küchlin, "Formal Methods for the Validation of Automotive Product Configuration Data," in *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 17(1), pp. 75-97, January 2003.

[13] J. Whittemore, J. Kim, and K. A. Sakallah, "SATIRE: A New Incremental Satisfiability Engine," in *Proc. 38th IEEE/ACM Design Automation Conference* (DAC), pp. 542-545, June 2001, Las Vegas, Nevada

[14] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 279-285, 2001.

[15] L. Zhang and S. Malik, "Extracting Small Unsatisfiable Cores from Unsatisfiable Boolean Formula," presented at Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), S. Margherita Ligure - Portofino, Italy, 2003.