From Max-SAT to Min-UNSAT: Insights and Applications

Mark H. Liffiton, Zaher S. Andraus, Karem A. Sakallah

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122

{liffiton, zandrawi, karem}@eecs.umich.edu

ABSTRACT

This report describes a strong connection between maximum satisfiability and minimally-unsatisfiable subformulas of any constraint system, as well as techniques for exploiting it. Focusing on CNF formulas, we explore this relationship and present novel algorithms for extracting minimally-unsatisfiable subformulas, including one that finds all such subformulas. We present experimental results showing how these can be used in counterexample-guided abstraction refinement for hardware verification, decreasing the number of iterations the process requires to verify a design. A large set of benchmarks is used to investigate the relationship between maximum satisfiability and minimally-unsatisfiable subformulas in a product configuration application.

1 INTRODUCTION

Many problems in hardware design and verification are posed as constraint satisfaction problems, most often in the form of Boolean CNF formulas analyzed with satisfiability (SAT) solvers. While SAT solvers can return a short proof in the form of an assignment when a formula is satisfiable, typically no proof or explanation is given when a formula is found to be unsatisfiable. Explanations of infeasibility are often valuable, and techniques for finding them have been developed for use in these problems. Some techniques have focused on reducing the original set of constraints to produce a minimal, unsatisfiable core representing a cause of infeasibility. In this report, we present a new approach to finding these cores, from a direction not yet explored in other work.

Consider an unsatisfiable CNF formula φ. A minimally-unsatisfiable subformula (MUS) of φ is an unsatisfiable subset of the clauses in φ such that removing any clause from the set makes it satisfiable. An MUS can be seen as an irreducible cause of the infeasibility of the original formula. φ could have multiple reasons for its infeasibility, and the removal of any one may not make φ satisfiable. In this case φ contains multiple MUSes. As long as any MUS is present in the formula, it will remain infeasible. In many applications, it is valuable to find the set of all MUSes.

We can look to counterexample-guided abstraction refinement [7] for a motivating example. Abstraction has been applied to model checking to greatly enhance its scalability; one can abstract the complete state space of a design and search for violations of required properties in the more tractable approximated state space. Because the abstraction has hidden some details of the design through over-approximation, a violation, in the form of a counterexample, may be spurious, not representing the behavior of the concrete design. In this case, the abstraction must be refined.

A counterexample can be applied to the concrete model and checked for consistency by way of a Boolean function that indicates whether the behavior could occur in the real design. If the formula is unsatisfiable, the counterexample represents a spurious behavior. An MUS of that formula points to elements of the abstraction that hid the real behavior, and these can be refined directly by adding constraints to the abstraction that prevent the behavior seen in the counterexample. The procedure then iterates by searching for a counterexample in the refined abstraction.

In many cases, the quality of the MUSes used, and thus the quality of the refinement, determines the number of iterations and the efficiency of the abstraction refinement process. Different types of abstraction have different criteria for optimal refinement. By finding all MUSes, one can perform high-level reasoning on them to determine which provide the best refinement. Alternatively, refining with all MUSes at once is guaranteed to subsume the optimal refinement, and it could additionally prevent spurious counterexamples not yet encountered. We present an example in Section 4.

Past work on minimally unsatisfiable subformulas has generally been either theoretical or experimental in nature. The theoretical side has focused mainly on the complexity of the problem, such as [17], which proves that the general problem of recognizing an MUS is DP-Complete, and [9,14,20], which place complexity bounds on recognizing MUSes with deficiency k (the deficiency of a formula is its number of clauses minus its number of variables).

Experimental work on finding MUSes can be grouped into algorithms that find a single MUS and those that find multiple, sometimes all, MUSes. Three algorithms from the former group are AMUSE [16], Bruni and Sassano's algorithm [4,5], and ZCore [22]. All of these utilize information from the resolution procedure of a modern SAT solver to find an unsatisfiable subformula, but they offer no guarantees on its minimality. The minimal unsatisfiability prover from [12] can complement these tools by removing unnecessary clauses from unsatisfiable subformulas to make them minimal. None of these techniques provide a reliable method of extracting more than one MUS.

One approach that does attempt to find an exact MUS and specifically searches for the *minimum* MUS (the smallest subset of unsatisfiable clauses) in a formula is presented in [15]. It employs a SAT solver to search through the entire set of unsatisfiable formulas to find the smallest one. In this way, it will find the solution exactly, but the technique is unlikely to scale well without further work, due to the extreme size of the search space. An algorithm with a similar search space, but intended to find all MUSes, is given in [3].

Others have noted a relationship between maximum feasibility and minimal infeasibility before (e.g., [4,6]). We have found a deeper connection, however, with many potential applications such as finding all MUSes of a formula. We describe this relationship and some of its uses in the sequel, which is organized as follows. In Section 2 we discuss the relationship between maximum satisfiability and minimally-unsatisfiable subformulas. The details of algorithms for exploiting this relationship are given in Section 3. Section 4 contains empirical results for an abstraction refinement application as well as a large set of automotive benchmarks. Finally, Section 5 ends the report with conclusions and potential future work.

2 THE RELATIONSHIP BETWEEN MAX-SAT AND MIN-UNSAT

The Maximum Satisfiability problem (Max-SAT) is an optimization problem on a CNF formula φ in which the goal is to find an assignment that maximizes the number of satisfied clauses. In other words, Max-SAT yields a satisfiable subset of φ's clauses with maximum cardinality. For example, $\varphi = (x_1) \wedge (\overline{x_1}) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2})$ has a Max-SAT solution with three satisfiable clauses: $\{(\overline{x_1}), (\overline{x_1} \vee x_2), (\overline{x_2})\}$.

Whereas the Max-SAT problem has been defined with the cardinality of a subset of clauses as the optimization goal, the problem can be relaxed to have *inaugmentability* as the goal instead. Instead of Max-SAT, defined as

$$\max\{|\psi| : \psi \subseteq \varphi,\ \psi \text{ is satisfiable}\}$$

we can define a new problem, which we will call *Maximally Satisfiable Subformula* (MSS) defined here, along with an analogous definition of the set of MUSes for comparison

$$\mathrm{MSS}(\varphi) = \left\{ \begin{array}{l} \eta : \eta \subseteq \varphi,\ \eta \text{ is satisfiable, and} \\ \forall c \in (\varphi \setminus \eta),\ \eta \cup \{\mathrm{c}\} \text{ is unsatisfiable} \end{array} \right\}$$

$$\mathrm{MUS}(\varphi) = \left\{ \begin{array}{l} \eta : \eta \subseteq \varphi,\ \eta \text{ is unsatisfiable, and} \\ \forall c \in \eta,\ \eta \setminus \{c\} \text{ is satisfiable} \end{array} \right\}$$

MSS(φ) is defined as the set of satisfiable subformulas of φ such that adding any remaining clause to one will render it unsatisfiable. Notice that MSS and MUS are essentially duals of one another! An MSS is satisfiable and cannot be made larger, and an MUS is *un*satisfiable and cannot be made *smaller*. Their relationship goes beyond this, though; we can use one set to quickly find elements of the other.

We will quickly point out a few properties of MSS. First, any Max-SAT solution is also an MSS solution; if a set is of maximum possible size, it cannot be made larger by definition. Solutions to MSS, however, may be of different sizes, and not all will necessarily have maximum cardinality. In the earlier example, $\{(\overline{x_1}), (\overline{x_1} \vee x_2), (\overline{x_2})\}$ is one set in MSS(φ), corresponding to the Max-SAT solution, and $\{(x_1), (\overline{x_2})\}$ is another, but of a different size.

Consider (x_1), the clause not included in the Max-SAT solution (which is also an MSS solution, as noted). It provides useful information; if it is removed from φ, the resulting formula is satisfiable. In general, the clauses not included in a maximally satisfiable subformula are a minimal-cardinality set of clauses that can be removed to make φ satisfiable. Therefore, we define "coMSS" to consist of these complementary sets; in a way, it contains the same information, but provides it in a different format:

$$\mathrm{coMSS}(\varphi) = \left\{\gamma : \gamma \subseteq \varphi,\ (\varphi \setminus \gamma) \in \mathit{MSS}(\varphi)\right\}$$

This complementary view of MSS reveals another connection between maximum satisfiability and minimally-unsatisfiable subformulas. Because the presence of any MUS in a formula φ makes φ unsatisfiable, at least one clause from every MUS in φ must be removed to make it satisfiable. That is, given an unsatisfiable formula φ and a set of clauses $\gamma \subseteq \varphi$, $\varphi' = \varphi \setminus \gamma$ is satisfiable if and only if γ contains at least one clause from every MUS in φ. Therefore, a set in coMSS(φ) must contain at least one clause from every MUS.

We can see that each set in coMSS provides an implicit solution to a covering problem on the set of MUSes (without being given the set explicitly or generating it at an intermediate step). Specifically, it is solving the HITTING-SET problem [10] on the set of MUSes. Each MUS is a subset of the clauses of φ, and a hitting set of the collection of MUSes is a set of clauses that contains at least one clause from every MUS. A set in coMSS is thus an irreducible hitting set of the set of MUSes. For convenience, we will refer to the sets in coMSS(φ) as *MUS covers* of φ. We will generally use γ to refer to an individual MUS cover.

In line with the "duality" of MSSes and MUSes, an MUS is also an irreducible hitting set of the set of MUS covers. We can give new definitions of the MUS covers

coMSS(φ)	A (x_1)	B (x_1')	C ($x_1' \vee x_2$)	D (x_2')
(x_1)	X			
(x_1'), ($x_1' \vee x_2$)		X	X	
(x_1'), (x_2')		X		X

MUS(φ)	A (x_1)	B (x_1')	C ($x_1' \vee x_2$)	D (x_2')
(x_1), (x_1')	X	X		
(x_1), ($x_1' \vee x_2$), (x_2')	X		X	X

$$\begin{aligned}\text{MUS}(\varphi) &= (\text{A})(\text{B} \vee \text{C})(\text{B} \vee \text{D}) \\ &= \text{AB} \vee \text{ACD} \\ &= \big\{\{(x_1),(x_1')\},\{(x_1),(x_1' \vee x_2),(x_2')\}\big\}\end{aligned}$$

$$\begin{aligned}\text{coMSS}(\varphi) &= (\text{A} \vee \text{B})(\text{A} \vee \text{C} \vee \text{D}) \\ &= \text{A} \vee \text{BC} \vee \text{BD} \\ &= \big\{\{(x_1)\},\{(x_1'),(x_1' \vee x_2)\},\{(x_1'),(x_2')\}\big\}\end{aligned}$$

Figure 1: Covering Problems Linking coMSS(φ) and MUS(φ)

and the MUSes, now written in terms of irreducible hitting sets of each other:

$$\text{coMSS}(\varphi) = \left\{\begin{array}{l}\gamma : \gamma \subseteq \varphi,\ \forall \mu \in \mathit{MUS}(\varphi),\ \gamma \cap \mu \neq \varnothing, \text{ and} \\ \forall c \in \gamma, \exists \mu \in \mathit{MUS}(\varphi).\ (\gamma \setminus \{c\}) \cap \mu = \varnothing\end{array}\right\}$$

$$\text{MUS}(\varphi) = \left\{\begin{array}{l}\mu : \mu \subseteq \varphi,\ \forall \gamma \in \Gamma(\varphi),\ \mu \cap \gamma \neq \varnothing, \text{ and} \\ \forall c \in \mu, \exists \gamma \in \Gamma(\varphi).\ (\mu \setminus \{c\}) \cap \gamma = \varnothing\end{array}\right\}$$

This states that an MUS cover γ of formula φ is a subset of the clauses in φ such that the intersection of γ with any MUS of φ is non-empty, and if any one clause is removed from γ, it will no longer have that property. Another way of stating the second half is to say that for any clause in γ there exists at least one MUS for which that clause is the only "representative" in γ. The definition of an MUS is equivalent to that of an MUS cover modulo exchanging MUS covers for MUSes.

Figure 1 illustrates this relationship with covering problems linking coMSS(φ) and MUS(φ) for the example formula given earlier. On the left is a table representing the covering problem that finds MUS(φ) from coMSS(φ). Each column is one clause from φ, and each row is an element of coMSS(φ). A clause covers a row if it is contained in the row's set of clauses (marked with an X in the table), and the goal is to select irreducible subsets of the clauses that cover all of the rows. This is done in the example in a manner similar to Petrick's Method for finding all minimum sum-of-products solutions from a prime implicant chart; each row becomes a disjunction of the columns that cover that row, and the disjunctions are conjoined and simplified by the distributive rule. The other half of the figure shows the same problem solved on the set of MUSes to find coMSS(φ).

Intuitively, we can see how the new definitions are equivalent to the first definitions given. With an earlier argument, we showed that the intersection of an MUS cover with any MUS is non-zero (i.e., each MUS cover contains at least one clause from every MUS), and this relationship is commutative. Both of the new definitions also state that their elements are irreducible, equivalent to saying that removing any element from one causes it to lose its defining property. This follows directly from the constraint of irreducibility in the earlier definitions.

With these new definitions, we have a complete implicit encoding of all MUSes within the collection of MUS covers, or a complete implicit encoding of the MUS covers if we have the collection of MUSes. They are essentially two sides of the same coin. In practice, it can be much easier to find maximally satisfiable subsets (and thus MUS covers) than to find minimally unsatisfiable subsets directly, so this relationship gives us a valuable bridge to reach MUSes more efficiently. If we can find the collection of MUS covers for a formula, we can obtain an MUS by finding an irreducible hitting set of the collection. In fact, we will show that an MUS can be extracted from the MUS covers in polynomial time.

This relationship between maximal feasibility and minimal infeasibility holds for any type of constraint system with hard constraints (i.e., constraints that are either satisfied or violated and can not be in any softer intermediate state). Therefore, anything related to this concept, including the following algorithms, can be applied to any type of constraint system with a well-defined concept of feasibility and a method for finding maximally feasible subsets of constraints.

3 ALGORITHM DETAILS

Exploiting this relationship to find all MUSes of a given CNF formula can be decomposed into two steps: 1) Finding the set of MUS covers coMSS(φ), and 2) Extracting MUSes from the set of covers. This decomposition is natural because the step of extracting MUSes cannot be done until the entire set of covers has been found, and the technique used to solve one subproblem does not affect the solution of the other.

3.1 Finding coMSS(φ)

As defined earlier, each MUS cover is the set of clauses *not* included in some maximally satisfiable subformula. To find all MUS covers, one must find all MSSes. One way to find an MSS is to solve the more constrained Max-SAT problem, and in our proof-of-concept implementation, we employ an incremental Max-SAT proce-

dure. Algorithm 1 provides a pseudocode outline of the procedure.

To provide a means of adding and removing clauses and enabling or disabling them within a constraint solver, every clause $C_i = x_i^1 \vee \ldots \vee x_i^m$ in φ is augmented with a negated *clause selector* variable y_i to give $C_i' = \overline{y}_i \vee x_i^1 \vee \ldots \vee x_i^m$ in a new formula φ'. While solving φ', assigning a certain y_i FALSE has the effect of disabling or removing C_i from the set of constraints, as the augmented clause is satisfied by the assignment to y_i. Conversely, assigning y_i TRUE enables the original clause. Max-SAT is solved by finding a satisfying assignment with a minimal number of y_i variables assigned FALSE, which ensures that as few constraints as possible are disabled. The clauses left unsatisfied, which are an MUS cover, are indicated by the set of y_i variables assigned FALSE in the optimal solution.

Instead of solving an optimization problem for every solution, however, we utilized a sliding objective approach. We set a bound on the number of y_i that may be assigned FALSE using an AtMost bound. Given a set of literals $\{l_1, l_2, \ldots, l_n\}$ and a positive integer k, an AtMost bound is defined as

$$\text{AtMost}\left(\{l_1, l_2, \ldots, l_n\}, k\right) \equiv \sum_{i=1}^{n} \text{assign}\left(l_i\right) \leq k$$

where assign(l_i) is 1 if l_i is assigned TRUE and 0 otherwise. In this application, then, we add a constraint of the form AtMost($\{\overline{y}_1, \overline{y}_2, \ldots, \overline{y}_n\}, k$), bounding the number of y_i variables assigned FALSE. For each bound, starting at 1 and increasing, we exhaustively search for all satisfiable assignments.

When one solution is found, the search continues incrementally after adding a blocking clause that forces out that solution. The blocking clause is a disjunction of the y variables for the clauses in the previous solution. For example, if the solution contains $y_2 = y_4 = y_7 = F$, indicating that clauses $\{C_2, C_4, C_7\}$ are an MUS cover, then adding $y_2 \vee y_4 \vee y_7$ to φ' will prevent finding that same MUS cover, or any superset of it, in any future solutions. Finding covers in order of increasing size and preventing supersets from being future solutions ensures that only irreducible MUS covers are found.

Adding a new constraint does not violate any of the learned clauses or other work done in solving the problem, so an incremental solver can be used. Each new solution is removed with a blocking clause and the search continues until no further solutions exist for the current bound. Incrementing the bound at this point relaxes a constraint on the system, so the search must start over with a new copy of the formula, augmented with all blocking clauses created thus far. (It may be possible to retain some learned clauses from the previous iteration, but we have not investigated this at this time.)

The search halts when φ' augmented with the collected blocking clauses is no longer satisfiable with no bound on the y_i variables. At this point, the blocking clauses will prevent the disabling of any clauses in any MUSes, so no satisfying assignment will exist. When this occurs, the entire set of MUS covers has been found.

Algorithm 1 (FindCovers) Finds all MUS covers of a given CNF formula.

```
findcovers(formula)
  // formula is a CNF instance
  AddYVars(formula)
  bound = 1
  Covers = null
  while (formula is satisfiable)
    formulaCopy = formula
    AddAtMost(formulaCopy, bound)
    while (formulacopy is satisfiable)
      newCover=incrementalSolve(formulaCopy)
      push(Covers, newCover)
      AddBlocking(formulaCopy, newCover)
      AddBlocking(formula, newCover)
    bound++
  return Covers
```

3.2 Obtaining MUS(φ)

The set of MUS covers implicitly encodes the entire set of MUSes of a formula, and information can be extracted from it in a variety of ways. Here, we will focus on methods for extracting MUSes, though it is likely that other useful data can be obtained by analyzing the set as well.

Extracting a Single MUS in Polynomial Time. Every MUS of a formula φ is an irreducible hitting set of the MUS covers of φ. Although MINIMAL-HITTING-SET is an NP-Complete problem [13], an irreducible hitting set for the set of MUS covers can be found in polynomial time, in part because of the fact that no cover is a subset of any other. With this condition, an MUS can be found by a straightforward iterative construction, with no search necessary. Algorithm 2 outlines the construction in pseudocode.

Intuitively, we want to generate a set of clauses, with at least one clause from each cover, such that every clause is an essential element of the set. By

Algorithm 2 (ExtractMUS) Generates one MUS from a set of MUS covers in polynomial time.

```
ExtractMUS(Covers)
  // Covers is the set of all MUS covers
  MUS = null
  while (Covers != null)
    curCover = pop(Covers)
    curClause = pop(curCover)
    push(MUS, curClause)
    for all testClause in curCover
      for all testCover in Covers
        if testCover contains testClause
          remove testClause from testCover
    for all testCover in Covers
      if testCover contains curClause
          remove testCover
  return MUS
```

"essential" we mean that removing a clause will leave at least one MUS cover unrepresented in the generated MUS; this enforces the irreducibility requirement.

The algorithm works by sequentially adding clauses to a forming MUS. When a clause is selected for inclusion, the remaining problem can be altered to force that clause to be required. Specifically, when a clause C_i is chosen and inserted into the MUS, all of the other clauses in one cover γ in which C_i appears are removed from the remaining problem. Then, C_i is required, as it will be the only element chosen from γ. Additionally, any covers in which C_i appears are removed, because they are now represented in the MUS. After these modifications are made, a new clause is selected from the resulting set of covers and the algorithm iterates. When no more covers remain, the constructed set of clauses is a complete, exact MUS.

Extracting All MUSes. Finding all MUSes involves searching for all irreducible hitting sets of the set of MUS covers. In general, this may be impractical due to the possibly exponential number of MUSes, but in many cases the result is tractable.

One technique for extracting the complete set of MUSes from the MUS covers uses the general form of the polynomial time algorithm for extracting a single MUS, additionally employing branching on two decisions. At the points where **`curCover`** and **`curClause`** are chosen by selecting arbitrary elements from **`Covers`** and **`curCover`** respectively, the program can instead branch on both the cover and the clause choices. Thus, every iteration of the loop will become two stages of branching. At each even stage there will be one branch for every possible cover and in each odd stage a new branch for every clause in the previously selected cover. Duplicate branches are possible, and in practice they seem quite common, so various pruning heuristics can be employed to reduce the space searched without missing any unique MUSes.

Extracting the Smallest MUS. In some applications, it is most useful to find the *smallest* MUS, in terms of its cardinality [15]. The smallest MUS of a formula is the smallest set of clauses that contains at least one clause from every MUS cover. This can be formulated as an instance of the minimum set covering problem [10], with the MUS covers as the row constraints and the clauses of φ as variables that cover the columns. Any set covering algorithm can be applied to the set of MUS covers; we have successfully employed a common branch-and-bound algorithm in our experiments. Generally, this takes less time to solve than finding the complete set of MUS covers in the first place. Another approach would be to use heuristics to guide the polynomial time MUS construction described above towards smaller MUSes. For example, always choosing the clause that is present in the largest number of remaining covers could produce smaller MUSes.

```
module equivalence(data,clk);
input [7:0] data;
input clk;

wire [7:0] a = data;

// specification
wire [7:0] spec1 = a+1;
wire [7:0] spec2 = a>>1;

// pipelined implementation
reg [7:0] impl1;
reg [7:0] impl2;
wire [7:0] arg = 8'd1;

initial begin
  impl1 = 0;
  impl2 = 0;
end
always @(posedge clk) begin
  impl1 = a+arg;
  impl2 = {1'b0,a[7:1]};
end

// equal==1 verifies the implementation matches
// the specification starting from cycle 1.
wire equal = spec1==impl1 && spec2==impl2;
endmodule
```

Figure 2: Equivalence Problem Verilog Model

4 EXPERIMENTAL RESULTS

The algorithm for finding MUS covers was implemented using MiniSAT [8] as a framework for constraint solving. MiniSAT can be extended with new types of constraints through an object-oriented interface in C++. This made it possible to integrate AtMost constraints alongside standard Boolean CNF clauses as needed by the algorithm. For extracting MUSes from the covers, the branching MUS construction algorithm was used, implemented in C++.

4.1 Abstraction Refinement

We integrated our implementation into the Vapor framework [1], which performs counterexample-guided abstraction refinement to verify Verilog designs. The abstraction models bit vectors with first-order logic terms and operators with uninterpreted functions. Verification yields abstract counterexamples, the feasibility of which can be checked by reduction to satisfiability of a Boolean formula involving the counterexample and the concrete model. If the formula $ConcreteModel \wedge Counterexample$ is unsatisfiable, the constraints given by the counterexample represent behavior inconsistent with that of the concrete model. In this case, the model is refined, using MUSes of that formula, to eliminate the spurious counterexample.

To demonstrate the effect of using multiple MUSes in one refinement step, we have used the integrated tool to verify an equivalence problem represented in Verilog; the Verilog model is shown in Figure 2. The tool produces a counterexample represented as a set of first-order logic constraints on the model's bit vectors. We are then able to find multiple MUSes of the resulting Boolean formula, each representing a subset of these constraints.

A subset of the constraints that, conjoined, form one such counterexample is shown here:

```
  3   shift_right(data,const1) = alpha1
  4
concat_7_1(sel_0_6(data),const0)=alpha2
  5   alpha1 != alpha2
  6   sel_0_6(data) = const1
  7   a = const0
  8   b = const1
  9   add(arg,const1) = alpha3
  10  f = arg+1
  11  alpha3 != f
```

In these constraints, `data` and `arg` are 8 bits wide, and the alphas are auxiliary bit vectors. This example has three representative MUSes. The set of constraints $mus_1 = \{3, 4, 5\}$ simplifies to the statement

```
(data >> 1) != {1'b0 , data[6:0]}
```

where `>>`, `{}`, and `[]` are the Verilog operators shift-right, concatenate, and extract, respectively. This constraint is unsatisfiable, and it is one cause of the false counterexample. Additional MUSes of the CNF formula represent the sets $mus_2 = \{6, 7\}$ and $mus_3 = \{9, 10, 11\}$. The abstract model can be refined, eliminating the false counterexample, by asserting the negation of each MUS.

This example shows two advantages of refining with multiple MUSes. First, all causes of a spurious counterexample are simultaneously removed, which reduces the number of needed refinement iterations dramatically. In our example, refining with mus_1 alone allows the possibility of mus_3 appearing in future iterations. The second advantage is demonstrated by mus_2. This MUS (along with several others) showed up in the counterexample, but it was not triggered directly by the Verilog code. Rather, it is an artifact of the high level of abstraction used. While checking the feasibility of the counterexample, the distinction between "real" and "side-effect" MUSes is not possible. In fact, refining with one MUS at a time could effectively waste refinement iterations removing many of these side-effects.

We performed the verification multiple times, each time changing the number of MUSes used in every refinement iteration. We found that using more MUSes at once did lead to fewer refinement iterations. Specifically, using one MUS at each step, or any fewer than five, led to more than 20 iterations. Once five MUSes were used in each refinement, the number of iterations dropped to 4. Finally, using 34 or more MUSes at once reduced the number of iterations to 3.

4.2 Automotive Benchmarks

To investigate the structure and properties of MUSes in a real-world application, we gathered data on a large set of unsatisfiable CNF benchmarks for automotive product configuration [18,19]. Each benchmark encodes a set of available configurations for a product, along with constraints enforcing a specific property to be checked. We observed that the encodings were not "tight," in that they contained numerous duplicate clauses. Duplicate clauses can yield a combinatorial explosion of MUSes, so they were removed before gathering data. There are a total of 84 benchmarks in the set, each with around 1500-1800 variables and 4000-8000 clauses. The data were collected in Linux on a PC with a 2.2GHz Opteron processor.

Due to space constraints, we cannot present the complete set of data here; instead, we highlight some representative results. Perhaps the most interesting result overall is the lack of consistency between benchmarks, even though they were encoded with the same techniques from one set of data. Some instances have hundreds of thousands of MUS covers, while others have very few. C202_FS_SZ_74, with 1,556 variables and 5,561 clauses, has 525,723 MUS covers, and C202_FS_SZ_121, with the same number of variables and 200 fewer clauses, has only 24. The number of MUSes also varies greatly. Twelve instances have one MUS, the size of which ranges from 31 to 213 clauses. One instance, C208_FA_UT_3255, has 52,736 MUSes, whose sizes were between 40 and 74 clauses. For others, the MUS extraction algorithm generated more than a million MUSes before reaching a twenty minute timeout in our experiments. By inspection of some instances with structurally simple MUS covers, we found that some of the instances have over 10^{39} distinct MUSes.

5 CONCLUSION AND FUTURE WORK

This report introduces a relationship between maximum satisfiability and minimally unsatisfiable subsets of constraints. The relationship is a rich one, and we have shown multiple techniques for deriving useful information using it. We have applied some of these techniques to a hardware verification process, showing how they can be used to increase its efficiency. Experimental results also indicate that there is much to be learned about minimal unsatisfiability using the relationship, with relevant properties varying greatly between similar CNF instances and no known good predictor for these properties. Further research on this topic can proceed in a number of areas:

- The connection between maximal satisfiability and minimal unsatisfiability we describe has been independently noted by Bailey and Stuckey [2], who applied it to type-error checking in software verification. Their implementation, however, differs from ours in several important areas. Most notably, they use a constraint solver as a subroutine, as opposed to fully integrating their search into the solver to automatically obtain benefits from its advanced heuristics and pruning techniques. Future work should include an experimental comparison of their approach, adapted to Boolean constraints, with our own.
- The process of finding MUS covers could be made more efficient with changes to either the implementation or the algorithm itself. The current implementation uses a general constraint solver, whereas algorithms specific to Max-SAT (e.g., [11,21]) could perform much better. The algorithm itself could be modified to exploit common structures

in CNF formulas such as symmetry or to use domain-specific knowledge. Performance may also be improved by relaxing optimality constraints and finding approximations of MUSes.

- There may be much more useful information to be obtained from the set of MUS covers. As an implicit encoding of the entire set of MUSes, it holds a great deal of information. We have shown three different analyses of these data, and others exist. Extracting only those MUSes that have certain useful properties is one possibility.
- Finally, we believe that the ideas presented here can be of use in learning more about the properties of MUSes and their relation to higher-level causes of infeasibility in constraint systems. Both the set of MUS covers and the complete set of MUSes (when tractable) could provide valuable information to developers of constraint systems and those investigating the "structure" of infeasibility.

ACKNOWLEDGMENTS

This work was funded in part by the National Science Foundation under ITR Grant No. 0205288.

REFERENCES

[1] Z. Andraus and K. Sakallah. "Automatic Abstraction and Verification of Verilog Models." In *Proc. of the 41st annual conference on Design automation* (DAC), pages 218-223, 2004.

[2] J. Bailey and P. J. Stuckey. "Discovery of Minimal Unsatisfiable Subsets of Constraints Using Hitting Set Dualization." In *Proc. of the 7th International Symposium on Practical Aspects of Declarative Languages* (PADL05), volume 3350 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[3] M. de la Banda, P. Stuckey, and J. Wazny. "Finding All Minimal Unsatisfiable Subsets." In *Proc. of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming* (PPDP 2003), pages 32-43, 2003.

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

[5] R. Bruni. "Approximating Minimal Unsatisfiable Subformulae by Means of Adaptive Core Search." *Discrete Applied Mathematics*, vol. 130(2), pages 85–100, 2003.

[6] J. W. Chinneck. "An effective polynomial-time heuristic for the minimum-cardinality IIS set-covering problem." *Annals of Mathematics and Artificial Intelligence*, vol. 17, pages 127 -144, 1996.

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-guided Abstraction Refinement." In *Computer Aided Verification*, pages 154-169, 2000.

[8] N. Eén, and N. Sörensson. "An Extensible SAT-solver." In *Sixth International Conference on Theory and Applications of Satisfiability Testing* (SAT03), 2003.

[9] H. Fleischner, O. Kullmann, and S. Szeider. "Polynomial-time recognition of minimal unsatisfiable formulas with fixed clause-variable difference." *Theoretical Computer Science*, vol. 289 no. 1, pages 503–516, 2002.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman*, San Francisco, 1979.

[11] S. de Givry, J. Larrosa, P. Meseguer, and T. Schiex. "Solving Max-SAT as weighted CSP." In *Principles and Practice of Constraint Programming* (CP2003), 2003.

[12] J. Huang. "MUP: A Minimal Unsatisfiability Prover." In *Proc. of the Tenth Asia and South Pacific Design Automation Conference* (ASP-DAC), January 2005.

[13] R. M. Karp. "Reducibility Among Combinatorial Problems." In *Proc. of a Symposium on the Complexity of Computer Computations*, pages 85-103, 1972.

[14] O. Kullmann. "An application of matroid theory to the SAT problem." In *Proc. of the 15th Annual IEEE Conference on Computational Complexity* (CCC2000), pages 116–124, 2000.

[15] J. Lynce and J. Marques-Silva. "On Computing Minimum Unsatisfiable Cores." In *Seventh International Conference on Theory and Applications of Satisfiability Testing* (SAT04), 2004.

[16] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. "AMUSE: A Minimally-Unsatisfiable Subformula Extractor." In *Proc. of the 41st Annual Conference on Design Automation*, pages 518–523, ACM Press, 2004.

[17] C. H. Papadimitriou and D. Wolfe. "The complexity of facets resolved." In *Journal of Computer and System Sciences*, vol. 37, pages 2–13, 1988.

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

[19] 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*, vol. 17 no. 1, pages 75-97, 2003.

[20] S. Szeider. "Minimal unsatisfiable formulas with bounded clause-variable difference are fixed-parameter tractable." *Journal of Computer and System Sciences*, vol. 69, no. 4, pages 656-674, 2004.

[21] H. Zhang, H. Shen, and F. Manyà. "Exact Algorithms for MAX-SAT." In *Electronic Notes in Theoretical Computer Science*, vol. 86 no. 1, 2003.

[22] L. Zhang and S. Malik. "Extracting small unsatisfiable cores from unsatisfiable Boolean formula." Presented at the Sixth International Conference on Theory and Applications of Satisfiability Testing, 2003.