MaxSAT-Based MCS Enumeration

Antonio Morgado[1], Mark Liffiton[2], and Joao Marques-Silva[1,3]

[1] CASL/CSI, University College Dublin, Dublin, Ireland
ajrm@ucd.ie
[2] Illinois Wesleyan University, Bloomington, IL, USA
mliffito@iwu.edu
[3] INESC-ID/IST, Lisbon, Portugal
jpms@ucd.ie

Abstract. Enumeration of *Minimal Correction Sets* (MCS) finds a wide range of practical applications, including the identification of Minimal Unsatisfiable Subsets (MUS) used in verifying the complex control logic of microprocessor designs (e.g. in the CEGAR loop of Reveal^{TM} [1,2]). Current state of the art MCS enumeration exploits core-guided MaxSAT algorithms, namely the so-called MSU3 [15] MaxSAT algorithm. Observe that a MaxSAT solution corresponds to a minimum sized MCS, but a formula may contain MCSes larger than those reported by a MaxSAT solution. These are obtained by enumerating all MaxSAT solutions. This paper proposes novel approaches for MCS enumeration, in the context of SMT, that exploit MaxSAT algorithms other than the MSU3 algorithm. Among other contributions, the paper proposes new blocking techniques that can be applied to different MCS enumeration algorithms. In addition, the paper conducts a comprehensive experimental evaluation of MCS enumeration algorithms, including both the existing and the novel algorithms. Problem instances from hardware verification, the SMT-LIB, and the MaxSAT Evaluation are considered in the experiments.

Keywords: AllMaxSAT, AllMaxSMT, MCS

1 Introduction

A Minimal Correction Subset (MCS) of an unsatisfiable CNF formula is an irreducible set of clauses whose removal causes the formula to become satisfiable (thus "correcting" it). MCSes can be naturally extended for Satisfiability Modulo Theories (SMT) formulas expressed in clausal form.

The enumeration of all Minimal Correction Subsets (MCSes) of unsatisfiable formulas finds a range of practical applications, including MUS enumeration [10] and design debugging [21]. One concrete example is the verification of hardware designs [2], for which enumeration of MCSes has been used in an industrial setting. A related problem is the enumeration of all *minimal* MCSes, those with the smallest cardinality. An example application is solving Boolean Multilevel Optimization by minimal MCS enumeration [3].

State of the art algorithms for MCS enumeration are based on model enumeration of Maximum Satisfiability (MaxSAT) solvers, and the most effective approaches are based on core-guided algorithms [11], more concretely the so-called MSU3 algorithm [14]. Nevertheless, in practical MaxSAT solving, the MSU3 algorithm is not as effective as other core-guided MaxSAT algorithms. Therefore, it is natural to ask how MCS enumeration can be extended to other MaxSAT algorithms. This question further motivates the investigation of different approaches for implementing model enumeration in MaxSAT algorithms.

This paper proposes improvements to MSU3-like MCS enumeration algorithms, and it shows how to implement MCS enumeration with other well-known MaxSAT algorithms, namely W(MSU1) [9,12]. Experimental results, obtained on a representative set of benchmarks, show that the proposed improvements are effective.

The remainder of the paper is organized as follows. Section 2 introduces the definitions and notation used throughout the paper. Afterwards, section 3 summarizes the application of MCS enumerations in verification with counterexample-guided abstraction refinement (CEGAR). Section 4 investigates improvements to existing MCS enumeration algorithms, and shows how other core-guided MaxSAT algorithms can be used for MCS enumeration. Experimental results are presented in Section 5, and the paper concludes in Section 6.

2 Preliminaries

This section provides basic definitions on SMT and MCSes and surveys some of the existing work on MaxSMT.

The problem of determining the satisfiability of a formula with respect to a background theory $\mathcal{T}$ is called the *Satisfiability Modulo Theory* (SMT) problem. Current SMT solvers are able to handle a variety of different theories and even conjunctions of theories. One example SMT theory is the theory of *Equality with Uninterpreted Functions* ($\mathcal{T}_e$), in which no restriction is imposed on the way the formulas or the predicates of a signature are interpreted.

Another example of a theory often seen in SMT instances is the theory of *Linear Integer Arithmetic* ($\mathcal{T}_{\mathcal{Z}}$), also know as the quantifier free *Presburger arithmetic*. Given the signature $(0, 1, +, -, \leq)$, $\mathcal{T}_{\mathcal{Z}}$ is the theory of models that interprets these symbols in the usual way over the integers [5]. Further details on SMT, theories and SMT solving can be obtained in [19,22,5].

This paper addresses the problems of finding all *Minimal Correction Sets* (MCSes) in SMT. Despite focusing on SMT, all the algorithms and techniques described in the paper can be applied in the SAT domain. Before presenting the definition of the enumeration problems, some notation is introduced.

Given an unsatisfiable constraint system φ, a minimal correction set M of φ is a set of constraints whose removal yields a satisfiable formula $\varphi' = \varphi - M$ ("correcting" the infeasibility) and that is minimal in the sense that adding any constraint from M back into φ' will make it unsatisfiable.

In the paper we refer to the MaxSMT problem [17]. The input of the MaxSMT problem is a CNF SMT formula φ, which is a conjunction of clauses. A clause is a disjunction of literals, where the literals are either atomic formulas or the negation of atomic formulas. The output of MaxSMT is an assignment $\mathcal{A}$ (consistent with $\mathcal{T}$) that minimize the number of falsified clauses of φ.

Generalizations of MaxSMT, include *Partial* MaxSMT, *Weighted* MaxSMT and *Weighted Partial* MaxSMT. In Partial MaxSMT the set of clauses in φ is divided in two separated sets: *hard* clauses and *soft* clauses. The goal is to minimize the number of soft clauses that are falsified while still satisfying all the hard clauses. Weighted MaxSMT allow weights on the clauses, with the objective of minimizing the sum of the weights of the falsified clauses, and Weighted Partial MaxSMT combines the previous two.

Two different enumeration problems are addressed in the paper and are defined as in Definition 1.

Definition 1. *Given a constraint system φ, the* AllMinMCS *problem consists of finding all the minimum size MCSes of φ. The* AllMCS *problem consists of finding all the MCSes of φ (independent of their size).*

Both AllMinMCS and AllMCS can be generalized to partial or/and weighted variants, analogous to MaxSMT. Observe that any MaxSMT solution indicates an MCS, in that the constraints not satisfied by that solution must be an MCS, and any such solution is a *smallest* MCS. The definition of an MCS requires minimality (not minimum cardinality), however, an instance can contain MCSes larger than those indicated by a MaxSMT solution, as well. Therefore, one can consider the problem of AllMinMCS to be similar to "AllMaxSMT", finding all MaxSMT solutions, and AllMCS is a somewhat broader problem.

The algorithms proposed in Section 4 are based on unsatisfiable cores. In SMT, as in the SAT domain, a *core* of an unsatisfiable CNF SMT formula φ is an unsatisfiable subset of clauses of φ. The SMT solver used in the experiments (Yices [8]) is capable of extracting cores from unsatisfiable instances.

2.1 MaxSMT

To the best of our knowledge, the first attempt to solve optimization problems using SMT (and in particular MaxSMT) was due to Nieuwenhuis & Oliveras [17]. This work extends the *Abstract DPLL Modulo Theories* [18] framework in order to be able to *strengthen* the theory. The strengthening of the theory allows the inclusion of new information (for example the improvement of a bound). Nieuwenhuis & Oliveras [17] applied a branch-and-bound schema using their framework for the case of weighted MaxSMT. Initially, each clause C_i (with a weight w_i) receives a new Boolean variable p_i, and the constraints $(p_i \rightarrow (k_i = w_i))$, and $(\neg p_i \rightarrow (k_i = 0))$ are added to the theory. Also the constraint $(k_1 + \ldots + k_m \leq B)$ is added to the theory together with the relation $(B < B_0)$ (where B_0 is an estimation of the initial cost). Each time a new cost B_j is found, the theory is strengthened by adding the relation $(B < B_j)$ to the theory.

Algorithm 1 The MSU3-SMT Algorithm (based on [15,14])

MSU3-SMT(φ)

```
1  φ_W ← φ                                          ▷ φ_W is the working formula
2  RV ← ∅                                           ▷ Set of relaxation variables
3  λ ← 0                                  ▷ Lower bound on true relaxation variables
4  while true
5        do (st, φ_C, A) ← SMT(φ_W ∪ Enc(Σ_{r∈RV} r ≤ λ))
6           ▷ "φ_C" is an unsat core if st is false
7           ▷ "A" is satisfying assignment if st is true
8           ▷ "Enc" encodes cardinality constraint
9           if st = true then return A           ▷ Solution to MaxSMT problem
10          if |RV| < |soft(φ)|
11             then for each ω ∈ φ_C ∩ soft(φ)
12                        do RV ← RV ∪ {r}        ▷ r is new relax. var. created
13                           ω_R ← ω ∪ {r}
14                           φ_W ← φ_W \ {ω} ∪ {ω_R}
15          λ ← λ + 1
16          ▷ [[ Additional code for enumeration of MCS inserted here ]]
```

In 2010, Cimatti et al. [6] proposed a new theory called the theory of *Costs* $\mathcal{C}$ that allows modeling multiple cost functions, and they developed a decision procedure for $\mathcal{C}$. Using the theory $\mathcal{C}$, Cimatti et al. [6] showed how to address the problem of minimizing the value of one cost function subject to the satisfaction of a SMT($\mathcal{T}$) formula, which they called the *Boolean Optimization Modulo Theory* (BOMT) problem. The optimization itself is obtained by linear search or binary search, asserting atoms of $\mathcal{C}$ that bound the cost, and using an incremental SMT solver. Cimatti et al. [6] encoded the weighted partial MaxSMT into BOMT by adding a new Boolean variable A_j^i to each soft clause. Then, the cost function is the sum of the weights of the soft clauses, whose variable A_j^i is assigned true.

Other work on MaxSMT algorithms includes [16,23]. The work in [16] addresses the concrete problem of Maximum Quartet Consistency, where an SMT solver is used as a black box, and optimizes a cost function either using linear or binary search binary. The work in [23] addresses optimization in SMT formulas when the variables in the cost function are rational.

An early MCS enumeration algorithm by Liffiton & Sakallah [10] followed an iterative approach, checking for MCSes of size 1, 2, etc. and blocking solutions as they were found. This algorithm was later extended to exploit unsatisfiable cores [11], closely following the approach of the MSU3 MaxSAT algorithm of Marques-Silva & Planes [15,14] but extending it to enumerate MCSes. The core-guided enumeration algorithm, generalized to SMT, is reviewed in detail in Section 4, while the SMT version of the MSU3 MaxSAT algorithm on which it is based is presented briefly here.

Algorithm 2 The FM-SMT Algorithm [9]

FM-SMT(φ)

```
1  φ_W ← φ                                          ▷ φ_W is the working formula
2  λ ← 0                                            ▷ Bound on the number of iterations
3  while true
4        do (st, φ_C, A) ← SMT(φ_W)
5           ▷ "φ_C" is an unsat core if st is false
6           ▷ "A" is a satisfying assignment if st is true
7           if st = true then return A              ▷ Solution to MaxSMT problem
8           if λ = |soft(φ)| then return false      ▷ No MaxSMT solution
9           λ ← λ + 1
10          RV ← ∅
11          for each ω ∈ φ_C, ω tagged as soft
12                do RV ← RV ∪ {r}                  ▷ r is a new relax. var. created
13                   ω_R ← ω ∪ {r}                  ▷ ω_R is tagged soft
14                   φ_W ← φ_W \ {ω} ∪ {ω_R}
15          if RV = ∅ then false                    ▷ No MaxSMT solution
16          φ_W ← φ_W ∪ Enc(∑_{r∈RV} r = 1)         ▷ Encodes card. const. tagged hard
```

Algorithm 1 presents the pseudo-code of the MSU3-SMT algorithm for MaxSMT. MSU3-SMT iteratively expands the *set* of relaxable clauses to encompass each extracted core while constraining the *number* of clauses that are allowed to be relaxed. The advantage of MSU3-SMT over other core-guided approaches is that it only adds a maximum of one relaxation variable per clause and one additional constraint to restrict the number of clauses relaxed.

Other MaxSAT algorithms can be extended to compute MaxSMT solutions as well. In particular, this paper considers the FM-SMT algorithm, based on the MaxSAT algorithm of Fu&Malik [9]. The approach taken by FM-SMT is to iteratively neutralize cores as they are found by adding fresh relaxation variables to the soft clauses of each core. Because the objective is to minimize the number of falsified clauses, the algorithm adds a constraint to allow relaxing exactly one clause in each core per iteration, thus allowing one of the clauses in the core to be falsified. The pseudo-code of FM-SMT is depicted in Algorithm 2.

This paper also considers a variation of the FM-SMT algorithm that uses an atMost1 constraint instead of the exactly1 cardinality constraint on line 16 of Algorithm 2. This algorithm is referred to as MSU1-SMT (similarly to the MSU1 MaxSAT algorithm [14,13]).

3 AllMCS in CEGAR

One direct application of the AllMCS problem, and one for which Section 5 contains empirical results, is verification via counterexample-guided abstraction refinement (CEGAR). Reveal$^{\text{TM}}$[1] is one such formal verification system that

Algorithm 3 The ALLMCS-MSU3-SMT Algorithm

```
ALLMCS-MSU3-SMT(φ)
1  φ_W ← φ
2  RV ← ∅
3  λ ← 0
4  while true
5        do (st, A) ← MSU3-SMT (φ, φ_W, RV, λ)
6           if st = true
7              then ReportMCS(φ, A)
8                   BlockMCSbyRV(φ_W, RV, A)
9              else exit
```

employs AllMCS in the process of verifying digital logic designs. Reveal performs datapath abstraction on a design and relies heavily on refinement, the dual of abstraction, to dynamically bridge between the abstract model and the original design throughout the verification process.

Specifically, when a violation is found in the abstract model, the flow produces a conjunction of bit-vector constraints representing the violation that indicate either a potential bug in the design or a "false alarm" resulting from the abstraction. Each violation must be checked against the original design to determine whether it is spurious, and this is done by checking the satisfiability of the violation's constraints V conjoined with the constraints of the original design C. If $V \wedge C$ is satisfiable, then the violation indicates a potential bug, and the flow exits with that result, but if $V \wedge C$ is UNSAT, then the violation is spurious and the abstraction must be refined. Each minimal subset of the violation $V' \subseteq V$ such that $V' \wedge C$ is UNSAT provides a concise reason for the contradiction in the form of a refinement core or lemma that can be used to refine the abstract model (by blocking it from future solutions).

It is here that AllMCS is applied. Every minimal unsatisfiable subset (MUS) of $V \wedge C$ indicates a minimal V' that can be used for refinement, and AllMCS is used in the first phase of the CAMUS algorithm for computing all MUSes of an unsatisfiable constraint system [10]. Extracting all MUSes is a core component of Reveal's algorithm during refinement, providing 1 to 4 orders of magnitude speedup in run-time compared with other refinement techniques as observed in academic benchmarks [2]. Efficient all-MUS extraction, and thus efficiently solving AllMCS is expected to be essential in practical abstraction/refinement-based implementations of formal verification on real-life designs.

4 All(Min)MCS Algorithms

This section develops new algorithms for AllMinMCS/AllMCS. The MaxSAT-based approach proposed in [10,11] is briefly reviewed first. Then, the new algorithms are detailed.

Algorithm 4 Additional code to include in Algorithm 1 for AllMCS

```
1  (st, φ_C) ← SMT(φ_W)                          ▷ φ_C is an unsat core if φ_W is unsat
2  if st = UNSAT
3     then if |RV| = |soft(φ)|
4          ▷ if all soft clauses are relaxed and φ_W is UNSAT,
5          ▷ then all MCSes have been found
6             then return false
7          if φ_C ∩ soft(φ) = ∅
8             then return false        ▷ nothing to relax; thus, no more solutions
```

The current state of the art approach for enumerating MCSes is due to Liffiton & Sakallah [11]. The algorithm enumerates MCSes in increasing order of size, essentially by solving MaxSAT iteratively, blocking each solution as it is found. The most recent, core-guided version is an extension of the MSU3 algorithm that follows this procedure.

As with MaxSAT algorithms, this MCS enumeration algorithm is easily extended to SMT, and the pseudo-code for the ALLMCS-MSU3-SMT algorithm is presented in Algorithm 3. In the pseudo-code, the function *ReportMCS*() reports the MCS found, and the function *BlocksMCSbyRV*() blocks the current MCS from reappearing by adding a blocking constraint to the working formula:

$$\varphi_W \leftarrow \varphi_W \ \cup \bigvee_{\mathcal{A}(r)=1,\ r \in RV} \neg r_\omega \tag{1}$$

Blocking MCSes in this way, creating a clause with the negation of the satisfiable relaxation variables, is referred to in the paper as *blocking by using relaxation variables*.

One requirement for the extension of MSU3-SMT to enumerating MCSes is a guarantee that the algorithm stops once it has found and blocked all the MCSes. This is done by adding the code shown in Algorithm 4 to line 16 in Algorithm 1, The motivation is that once all MCSes have been blocked, calling the SMT solver without cardinality constraints will return false.

Observe that, the ALLMCS-MSU3-SMT algorithm always reports the MCSes in increasing size, because it iteratively asks for a MaxSMT solution on the current φ_W. As such, the same algorithm can be used to solve AllMinMCS by additionally stopping if the size of a newly found MCS is larger than the previous.

MSU3-SMT is based on the MSU3 MaxSAT algorithm of Marques-Silva & Planes [15,14]. MSU3 is a *core*-guided MaxSAT algorithm (once that it relies on unsatisfiable cores) that iteratively improves a lower bound. In the MaxSAT domain, another core-guided algorithm that also improve a lower bound is the FM MaxSAT algorithm of Fu & Malik [9]. For MaxSAT, and for some industrial applications, the FM algorithm performs better than MSU3. Indeed, in recent

MaxSAT Evaluations[4], the algorithms that follow the approach of Fu & Malik [9] abort on fewer instances than MSU3 in the MS-Industrial and in the WPMS-Industrial categories. Moreover, section 2 describes how to use the FM MaxSAT algorithm to create the FM-SMT algorithm.

Following the approach of Liffiton & Sakallah for MCS enumeration, enumerating MCSes using the FM-SMT algorithm corresponds to iteratively asking the FM-SMT solver for a MaxSMT solution and blocking each until no more solutions can be found. Since FM-SMT also uses relaxation variables, then the blocking of MCSes by using relaxation variables can be considered. The algorithm would be similar to ALLMCS-MSU3-SMT but using FM-SMT instead of MSU3-SMT. Nevertheless, using FM-SMT to enumerate MCSes presents some additional complications.

One problem that arises with this approach is the termination of the algorithm. The original FM MaxSAT algorithm *does not include* the check done in line 8 of Algorithm 2. Suppose that FM-SMT does not make the check in line 8 and that the underlying SMT solver always returns as a core the full formula (that is $\varphi_C = \varphi_W$). Then after blocking all the MCSes, the FM-SMT algorithm should be able to report that the formula does not have any MaxSMT solution and exit on line 15 of the FM-SMT algorithm. Nevertheless, since the core obtained is the full formula, then the algorithm is always able to add new relaxation variables (line 12), and it enters a new loop where it continues with new relaxation variables.

Thus, in order to guarantee that enumerating MCSes with FM-SMT always terminates, the check done in line 8 of Algorithm 2 has to be performed. In the original FM MaxSAT algorithm (or also for solving MaxSMT with FM-SMT), this problem does not arise, since in MaxSAT (MaxSMT) the algorithm returns after finding the first solution (and not blocking it as in enumeration).

Another problem that arises with the FM-SMT algorithm for enumeration is the presence of duplicates and supersets of MCSes (which are not themselves MCSes) as shown in Example 1.

Example 1. Consider for example the following CNF SMT formula with 5 soft clauses:

$$\varphi = \{(x \geq 1),\ (x < 1),\ ((x < 1) \vee (y < 1)),\ (y < 1),\ (y \geq 1)\}$$

On the first call to the FM-SMT algorithm, the solver finds two cores before returning a solution. Suppose the cores founds are $\varphi_C^1 = \{(x \geq 1), (x < 1)\}$ and $\varphi_C^2 = \{(y \geq 1), (y < 1)\}$. The algorithm updates λ twice (that is $\lambda = 2$) and the working formula to:

$$\begin{aligned}\varphi_W = \{&((x \geq 1) \vee r_1),\ ((x < 1) \vee r_2),\ ((x < 1) \vee (y < 1)),\\ &((y < 1) \vee r_3),\ ((y \geq 1) \vee r_4)\} \cup\\ &\mathrm{Enc}(r_1 + r_2 = 1) \cup\\ &\mathrm{Enc}(r_3 + r_4 = 1)\end{aligned}$$

[4] MaxSAT Evaluations, `http://www.maxsat.udl.cat`.

Suppose the solution reported is such that $\mathcal{A}(r_1) = \mathcal{A}(r_4) = 1$ and $\mathcal{A}(r_2) = \mathcal{A}(r_3) = 0$, then the enumerating algorithm will report the MCS $\{(x \geq 1),\ (y \geq 1)\}$ and add the blocking constraint $(\neg r_1 \vee \neg r_4)$ to the working formula.

In the next two iterations of the enumerating algorithm, the FM-SMT algorithm will always report a MaxSMT solution without adding any relaxation variable, and the enumerating algorithm will report the two MCSes $\{(x \geq 1),\ (y < 1)\}$ and $\{(x < 1),\ (y \geq 1)\}$, which after blocking the MCSes the working formula is as follows:

$$\begin{aligned}\varphi_W = \{&((x \geq 1) \vee r_1),\ ((x < 1) \vee r_2),\ ((x < 1) \vee (y < 1)),\\ &((y < 1) \vee r_3),\ ((y \geq 1) \vee r_4)\} \cup\\ &\text{Enc}(r_1 + r_2 = 1) \cup\\ &\text{Enc}(r_3 + r_4 = 1) \cup\\ &\{(\neg r_1 \vee \neg r_4)\} \ \cup\ \{(\neg r_1 \vee \neg r_3)\} \ \cup \{(\neg r_2 \vee \neg r_4)\}\end{aligned}$$

Now φ_W is unsatisfiable and the core returned by the SMT solver contains all the clauses. The FM-SMT algorithm adds fresh relaxation variables to each of the soft clauses and a new constraint on the new relaxation variables. Also λ is updated to 3. The resulting working formula is as follows:

$$\begin{aligned}\varphi_W = \{&((x \geq 1) \vee r_1 \vee r_5),\ ((x < 1) \vee r_2 \vee r_6),\ ((x < 1) \vee (y < 1) \vee r_7),\\ &((y < 1) \vee r_3 \vee r_8),\ ((y \geq 1) \vee r_4 \vee r_9)\} \cup\\ &\text{Enc}(r_1 + r_2 = 1) \cup\\ &\text{Enc}(r_3 + r_4 = 1) \cup\\ &\text{Enc}(r_5 + r_6 + r_7 + r_8 + r_9 = 1) \cup\\ &\{(\neg r_1 \vee \neg r_4)\} \ \cup\ \{(\neg r_1 \vee \neg r_3)\} \ \cup \{(\neg r_2 \vee \neg r_4)\}\end{aligned}$$

The current working formula is satisfiable and one of the solutions of φ_W is such that $\mathcal{A}(r_2) = \mathcal{A}(r_3) = \mathcal{A}(r_9) = 1$ and all the other relaxation variables are assigned 0. This MaxSMT solution would make the enumeration algorithm to report $\{(x < 1),\ (y < 1),\ (y \geq 1)\}$ as an MCS, which is wrong, because this set corresponds to a superset of a previous MCS.

The previous example shows the necessity of removing supersets and duplicated MCSes that may arise when enumerating MCSes with the FM-SMT algorithm. The pseudo-code of ALLMCS-FM-SMT is shown in Algorithm 5. In the pseudo-code, the function *isSuperSet*() obtains the current MCS and checks if it corresponds to a superset of a previous MCS, in which case it returns true. As such, an MCS is only reported as an MCS if it is not a superset of a previous MCS.

The problem of enumerating MCSes with the FM-SMT algorithm is that it may add relaxation variables to the same clause more than once. When relaxing a clause that has previously participated in a MCS that has been blocked, then it allows for the clause to re-occur in a new MaxSMT solution using the new relaxation variable.

The next section presents two new blocking techniques that do not require the enumeration of MCSes with FM-SMT to check for supersets or duplicates.

Algorithm 5 The ALLMCS-FM-SMT Algorithm

```
ALLMCS-FM-SMT(φ)
1  φ_W ← φ
2  λ ← 0
3  while true
4        do (st, A) ← FM-SMT (φ, φ_W, λ)
5           if st = true
6              then if (!isSuperSet(φ, A))
7                       then ReportMCS(φ, A)
8                    BlockMCSbyRV(φ_W, A)
9              else  exit
```

4.1 New Techniques for Blocking MCSes

This section proposes two new techniques for blocking MCSes. The motivation is to eliminate the need to use relaxation variables for blocking MCSes, as is done with *Blocking by using Relaxation Variables*. With these new techniques, MCSes will remain "blocked" independently of the way the MaxSMT solver manipulates the relaxation variables.

The first technique is inspired by the relaxation variables and is called *Blocking by using Auxiliary Variables*. Blocking by using auxiliary variables consists of initially transforming each soft clause into a hard clause after adding a fresh Boolean variable called an *auxiliary variable*. Additionally, a set of unit soft clauses is added that corresponds to the negation of each auxiliary variable. Consider the previous formula of Example 1, and suppose that blocking by using auxiliary variables is to be used. Then the formula given to the MaxSMT solver is the formula containing the set of soft clauses:

$$\varphi^{soft} = \{(\neg a_1),\ (\neg a_2),\ (\neg a_3),\ (\neg a_4),\ (\neg a_5)\}$$

and the set of hard clauses:

$$\varphi^{hard} = \{((x \geq 1) \vee a_1),\ ((x < 1) \vee a_2),\ ((x < 1) \vee (y < 1) \vee a_3),\\ ((y < 1) \vee a_4),\ ((y \geq 1) \vee a_5)\}$$

When the enumeration solver calls the function to block an MCS, then the blocking constraint to add to the working formula is as in the following Equation 2, where a_i are auxiliary variables.

$$\varphi_W \leftarrow \varphi_W\ \cup \{(\bigvee_{\mathcal{A}(a_i)=1} \neg a_i)\} \tag{2}$$

The second technique proposed does not require the addition of extra Boolean variables or the transformation of soft clauses into hard clauses. Instead, in *Blocking by using Original Literals*, the original literals in the clauses are used

for blocking the MCSes. Consider once more the previous formula of Example 1 and suppose that blocking by using original literals is being used for blocking MCSes. Suppose the algorithm has just found the MCS $\{(x < 1), ((x < 1) \vee (y < 1)), (y < 1)\}$, then the blocking clause added to the working formula is the clause $((x < 1) \vee (y < 1))$. In the general case, when the enumeration solver calls the function to block an MCS, then the blocking constraint to add to the working formula is as in the following Equation 3, where $(l_{i_1} \vee \ldots \vee l_{i_j})$ is the original literals in a clause that belongs to the current MCS found.

$$\varphi_W \leftarrow \varphi_W \ \cup \{(\bigvee_{\mathcal{A}(l_{i_1} \vee \ldots \vee l_{i_j})=0} l_{i_1} \vee \ldots \vee l_{i_j})\} \tag{3}$$

Note that since these two techniques deal directly with the MCSes, and not with the relaxation variables, then enumeration with the FM-SMT algorithm using these techniques will not report supersets of previous MCSes and as such does not require the check if the reported MCS is a superset of a previous MCS.

Observe that these two new blocking techniques can be applied not only in enumeration with the FM-SMT algorithm (and the MSU1-SMT algorithm) but also with the MSU3-SMT algorithm.

4.2 AllMCS with Costs

The AllMCS problem can be extended to weighted variants of MaxSAT/MaxSMT. Namely, each soft constraint can be associated with a weight. This weight represents the cost of adding the constraint to a MCS, i.e. of not satisying the clause.

The goal is then to enumerate all of the MCSes taking into account the sums of the weights of their clauses. Two approaches can be considered. The first approach consists of extending the AllMCS algorithms to handle weights. For example, this can be done by using a weighted MaxSMT solver with an AllMCS algorithm. However, recent results from the MaxSAT Evaluations confirm that weighted MaxSAT is in practice harder to solve than non-weighted MaxSAT, and confirms the different complexity classes of these problems.

Nevertheless, a simpler solution exists. Observe that the MCSes of a CNF formula are *independent* of the weights associated with the clauses. That is, an MCS of a weighted CNF formula is also an MCS of the corresponding unweighted formula and vice-versa. Thus, for the case of weighted formulas, it suffices to use one of the unweighted AllMCS algorithms outlined in earlier sections. Afterwards, one just needs to sort the MCSes by increasing (or decreasing) weight.

5 Experimental Results

This section presents a complete experimental evaluation of the enumeration algorithms described in earlier sections. In what follows, the instances used in the experiments are described, along with the experimental setup.

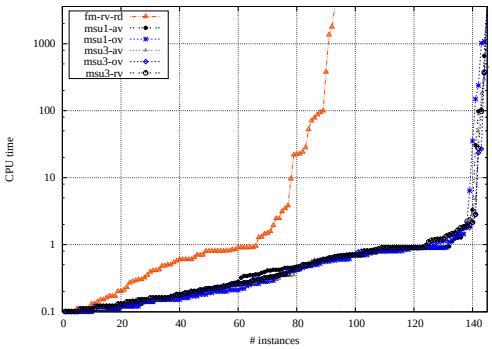

Fig. 1: Cactus plot for Reveal instances

	#Sol.	Sum NA (91)	Sum NA (143)
fm-rv-rd	92	2823.02	–
msu1-av	144	**146.72**	**302.13**
msu1-ov	144	285.35	1575.03
msu3-av	**145**	352	554.67
msu3-ov	144	410.48	455.79
msu3-rv	144	430.38	550.65

Table 1: Statistics for Reveal instances

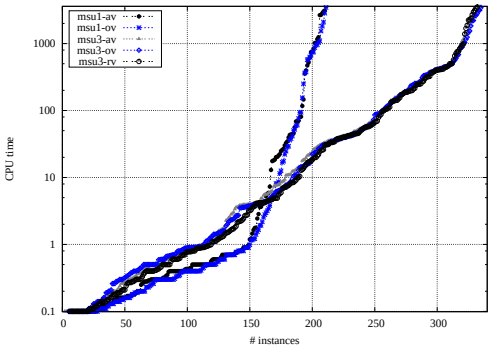

Fig. 2: Cactus plot for non-weighted MaxSAT instances

	#Sol.	Sum NA (183)
msu1-av	210	7495.95
msu1-ov	211	**6981.32**
msu3-av	**335**	8457.76
msu3-ov	**335**	7276.02
msu3-rv	332	7467.42

Table 2: Statistics for non-weighted MaxSAT industrial instances

Three classes of instances have been used in this work. The *Reveal* instances were generated in the Reveal digital logic verification flow as described in Section 3. These instances are characterized by having a single hard clause, with all other clauses being soft. A total of 145 unsatisfiable instances were obtained from Reveal Design Automation, Inc. The second class of instances is referred to as MaxSAT, and it consists of all *industrial* instances from the MaxSAT Evaluations from 2009 to 2011. Both weighted and unweighted industrial instances are considered, making a total of 1323 instances, where 12 instances are satisfiable. The instances were considered in two sets, weighted and non-weighted, giving 233 weighted instances (6 satisfiable) and 1090 non-weighted instances (6 satisfiable). The last class of instances considered, referred to as *SMT-LIB*, was obtained from the SMT-LIB [4], a library of SMT benchmarks developed for testing and validating SMT algorithms. The instances selected are in the SMT-LIB 1.2 format and belong to one of the following logics: QF_IDL, QF_LIA, QF_LRA, QF_RDL, QF_UF, QF_UFIDL, QF_UFLIA, QF_UFLRA, QF_UF. All clauses in this class of instances are considered *soft*.

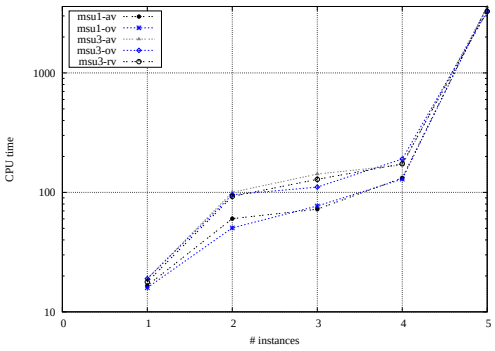

Fig. 3: Cactus plot for weighted MaxSAT instances

	#Sol.	Sum NA (4)
msu1-av	4	281.52
msu1-ov	4	**272.5**
msu3-av	**5**	431.2
msu3-ov	**5**	415.84
msu3-rv	**5**	412.54

Table 3: Statistics for weighted MaxSAT industrial instances

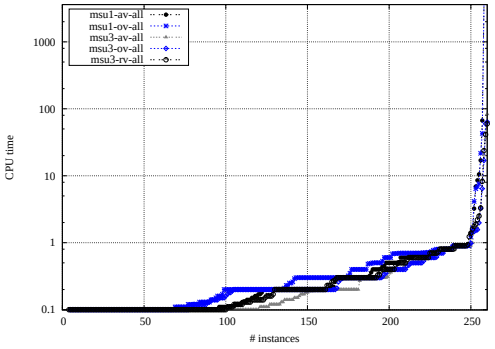

Fig. 4: Cactus plot of SMTLIB instances

	#Sol.	Sum NA (257)
msu1-av	257	184.29
msu1-ov	257	**171.2**
msu3-av	**260**	214.78
msu3-ov	**260**	219.4
msu3-rv	**260**	208.83

Table 4: Statistics about the results with SMTLIB instances

Given the large number of instances obtained from SMTLIB, a smaller subset was selected for the experiments. Each instance was given to the MSU1-SMT algorithm, and the instances for which MSU1-SMT reported a MaxSAT solution were then ordered by CPU time used. The first 808 instances in that order were selected for these experiments.

For the experimental results, both FM-SMT and MSU3-SMT have been implemented on top of yices [8]. We have also implemented the variant of FM-SMT that uses atMost1 constraints, referred to as MSU1-SMT.

In the experiments, all algorithms were configured to enumerate the complete set of MCSes. The algorithms make use of cardinality constraints, and in these experiments, the pairwise cardinality network encoding of Codish & Zazon-Ivry [7] was used for encoding each of the cardinality constraints into Boolean CNF. The only exception is for the FM-SMT algorithm, which uses the bitwise encoding [20], as it provided better results for this algorithm.

All the three techniques described in Section 4) for blocking one MCS (blocking by using *relaxation variables*, blocking by using *auxiliary variables* and block-

ing by using *original literals*), have been considered in the experiments, depending on the underlying MaxSMT algorithm used. For the ALLMCS-FM-SMT algorithm, only the blocking by using relaxation variables technique was used. Note that this is the only algorithm in the experiments which requires a verification of duplicated MCSes. The resulting algorithm is referred to as *fm-rv-rd*, and it was only tested with the Reveal instances.

For the ALLMCS-MSU1-SMT algorithm, both blocking by using auxiliary variables and blocking by using original literals were considered for all instances. The resulting algorithms are referred to as *msu1-av* and *msu1-ov*, respectively. For the ALLMCS-MSU3-SMT algorithm, all the blocking techniques were tried, and the algorithms obtained are referred to as *msu3-av*, *msu3-ov* and *msu3-rv*. Observe that msu3-rv corresponds to the approach of Liffiton & Sakallah [11].

The algorithms were run on a cluster of Intel Xeon E5450 (3 GHz) nodes running RedHat Linux v.5 x86-64 with a timeout of 3600 seconds and a memory limit of 4GB.

Figure 1 shows a cactus plot obtained from the Reveal instances, while Table 1 shows a summary of the results for each of the algorithms. In the table, *#Sol.* represents the number of instances solved by each of the algorithms, *Sum NA (91)* represents the sum of cputimes taken by each algorithm on 91 instances for which all of the algorithms finished within the timeout. Finally, *Sum NA (143)* represents the sum of cputimes taken by each algorithm on 143 instances which were solved by all algorithms except for fm-rv-rd. As can be seen from the cactus plot and from the number of solved instances in Table 1, the fm-rv-rd is the worst performing algorithm, aborting in more instances and requiring more cputime to enumerate even on the 91 instances solved by all the algorithms. This is due to the need to remove duplicated MCSes and supersets of current MCSes.

The algorithm that solves the largest number of the Reveal instances is msu3-av, able to solve one more instance than the other msu algorithms. Considering all the 143 instances where none of the msu algorithms aborts, msu1-av is the fastest enumerating algorithm.

For the non-weighted industrialMaxSAT instances, Figure 2 shows the cactus plot, while Table 2 summarizes the results for this class of instances. As before, column *#Sol.* shows the number of instances solved by each of the algorithms, and *Sum NA (183)* represents the sum of cputimes taken by each algorithm on 183 instances that were solved by all algorithms. From the cactus plot, it can be seen that overall, the msu3 algorithms perform better than the msu1 algorithms in this class of instances. Table 2 confirms that msu3 algorithms abort on fewer instances, and among the msu3 algorithms, msu3-av and msu3-ov are able to solve 3 more instances than msu3-rv. For the sum of cputimes on instances solved by all the algorithms, the fastest algorithm for these instances is msu1-ov, as the msu1 algorithms tend to be faster on the easier instances solved by all algorithms.

The results with weighted instances are presented in a the cactus plot of Figure 3 and summarized in Table 3. Despite these instances being weighted, the algorithms disregard the weights as suggested in Section 4.2. Column *#Sol*

show the same type of result as the in previous tables. Column *Sum NA (4)* represents the sum of cputimes taken by each algorithm on 4 instances which were solved by all algorithms. From the figure and table, it can be seen that msu3 algorithms actually solve one more instance than msu1 algorithms but, for the 4 instances solved by all, msu1 algorithms are faster than msu3 algorithms, where msu1-ov is the fastest algorithm (over these 4 instances).

The last plot, Figure 4, shows the cactus plot with the results obtained from the SMTLIB instances, while Table 4 summarizes. For these instances, the msu3 algorithms are able to solve 3 more instances than the msu1 algorithms. Nevertheless, considering only the 257 instances solved by all the algorithms, the fastest algorithm is msu1-ov.

Overall, it can be seen from the results that the msu3 algorithms solve the greatest number of instances, and in particular, msu3-av is the only algorithm that solved all of the Reveal instances. On the other hand, when considering only instances that are solved by all algorithms, the results suggest that the fastest algorithms are the msu1 algorithms, indicating that the easier instances are solved more quickly by msu1 algorithms than msu3 variants. On these, msu1-av is the fastest for the reveal instances, while for the other three classes, the fastest algorithm is msu1-ov. There is no substantial difference between the different blocking techniques when applied to a given algorithm (msu1 or msu3).

6 Conclusions and Future Work

State of the art algorithms for MCS enumeration of SAT and SMT instances are based on one concrete instantiation of core-guided MaxSAT algorithms [11]. This paper proposes improvements to MCS enumeration algorithms, and shows how these algorithms can integrate other core-guided MaxSAT algorithms. Experimental results, obtained on a wide range of practical instances of SAT and SMT, show that the proposed improvements reduce overall running times and allow solving more problem instances.

The proposed algorithms have been implemented on top of SMT solvers, using available interfaces. Direct access to the internal state of the SMT solver is expected to allow further performance improvements.

References

1. Andraus, Z.S.: Automatic Formal Verification of Control Logic in Hardware Designs. PhD Dissertation, University of Michigan (2009)
2. Andraus, Z.S., Liffiton, M.H., Sakallah, K.A.: Reveal: A formal verification tool for verilog designs. In: Logic for Programming Artificial Intelligence and Reasoning (LPAR-2008). pp. 343–352 (November 2008)
3. Argelich, J., Lynce, I., Marques-Silva, J.: On solving Boolean multilevel optimization problems. In: International Joint Conference on Artificial Intelligence. pp. 393–398 (2009)
4. Barrett, C., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2010)

5. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, pp. 825–885. IOS Press (2009)
6. Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R., Stenico, C.: Satisfiability modulo the theory of costs: Foundations and applications. In: Tools and Algorithms for the Construction and Analysis of Systems. vol. 6015, pp. 99–113 (2010)
7. Codish, M., Zazon-Ivry, M.: Pairwise cardinality networks. In: Logic for Programming Artificial Intelligence and Reasoning 2009 (LPAR-16) (2010)
8. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for dpll(t). In: Computer-Aided Verification. vol. 4144, pp. 81–94 (2006)
9. Fu, Z., Malik, S.: On solving the partial MAX-SAT problem. In: Theory and Applications of Satisfiability Testing. pp. 252–265 (2006)
10. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. Journal of Automated Reasoning 40(1), 1–33 (January 2008)
11. Liffiton, M.H., Sakallah, K.A.: Generalizing core-guided Max-SAT. In: Theory and Applications of Satisfiability Testing. pp. 481–494 (2009)
12. Manquinho, V., Marques-Silva, J., Planes, J.: Algorithms for weighted Boolean optimization. In: Theory and Applications of Satisfiability Testing. pp. 495–508 (2009)
13. Marques-Silva, J., Manquinho, V.: Towards more effective unsatisfiability-based maximum satisfiability algorithms. In: Theory and Applications of Satisfiability Testing. pp. 225–230 (2008)
14. Marques-Silva, J., Planes, J.: On using unsatisfiability for solving maximum satisfiability. Computing Research Repository abs/0712.0097 (2007)
15. Marques-Silva, J., Planes, J.: Algorithms for maximum satisfiability using unsatisfiable cores. In: Design, Automation and Test in Europe. pp. 408–413 (2008)
16. Morgado, A., Marques-Silva, J.: Combinatorial optimization solutions for the maximum quartet consistency problem. Fundam. Inform. 102(3-4), 363–389 (2010)
17. Nieuwenhuis, R., Oliveras, A.: On SAT modulo theories and optimization problems. In: Theory and Applications of Satisfiability Testing. pp. 156–169 (2006)
18. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Abstract DPLL and abstract DPLL modulo theories. In: Logic for Programming, Artificial Intelligence, and Reasoning. vol. 3452, pp. 36–50 (2004)
19. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(). Journal of the ACM 53(6), 937–977 (2006)
20. Prestwich, S.: Variable dependency in local search: Prevention is better than cure. In: International Conference on Theory and Applications of Satisfiability Testingnational Conference on Theory and Applications of Satisfiability Testing. pp. 107–120 (2007)
21. Safarpour, S., Mangassarian, H., Veneris, A., Liffiton, M.H., Sakallah, K.A.: Improved design debugging using maximum satisfiability. In: Formal Methods in Computer-Aided Design (November 2007)
22. Sebastiani, R.: Lazy satisfiability modulo theories. Journal on Satisfiability, Boolean Modeling and Computation 3(3), 141–224 (2007)
23. Sebastiani, R., Tomasi, S.: Optimization in SMT with $\mathcal{LA}(\mathbb{Q})$ cost functions. In: International Joint Conference Automated Reasoning. pp. 484–498 (2012)