Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)

Core-Guided Minimal Correction Set and Core Enumeration

Nina Narodytska[1], **Nikolaj Bjørner**[2], **Maria-Cristina Marinescu**[3], **Mooly Sagiv**[1,4]
[1] VMware Research, USA
[2] Microsoft Research, USA
[3] Barcelona Supercomputing Center, Spain
[4] Tel Aviv University, Israel
nnarodytska@vmware.com, nbjorner@microsoft.com, maria.marinescu@bsc.es, msagiv@acm.org

Abstract

A set of constraints is unsatisfiable if there is no solution that satisfies these constraints. To analyse unsatisfiable problems, the user needs to understand where inconsistencies come from and how they can be repaired. Minimal unsatisfiable cores and correction sets are important subsets of constraints that enable such analysis. In this work, we propose a new algorithm for extracting minimal unsatisfiable cores and correction sets simultaneously. Building on top of the relaxation and strengthening framework, we introduce novel techniques for extracting these sets. Our new solver significantly outperforms several state of the art algorithms on common benchmarks when it comes to extracting correction sets and compares favorably on core extraction.

1 Introduction

A set of constraints is over-constrained if there is no solution that satisfies these constraints. Over-constrained problems often occur in practical applications, including verification, configuration, diagnosis, etc., [Belov *et al.*, 2012; Liffiton and Sakallah, 2008b; Junker, 2004]. For an over-constrained problem, it is important to understand why the problem's constraints are inconsistent and how we can relax constraints to eliminate these inconsistencies. To do so, two informative notions were introduced in the literature [Reiter, 1987; Liffiton and Sakallah, 2008a; Liffiton *et al.*, 2016]. The first notion is an unsatisfiable core. An unsatisfiable core is a subset of constraints that is unsatisfiable regardless of the rest of the formula. We are interested in minimal unsatisfiable cores (MUSes) under the set inclusion relation as they represent a concise description of the problem's inconsistencies. Another important notion is a correction set, which is a subset of clauses removal of which makes the remaining formula satisfiable. Again, minimal correction sets (MCSes) are interesting for a user as we want to satisfy as many constraints as possible in the remaining formula. As minimal cores and correction sets are not unique, it is important to build tools and algorithms to enumerate them. Applications in verification and knowledge representation rely on the ability to enumerate these sets [Liffiton and Sakallah, 2008b].

Several new algorithms have been developed during the past decade for efficient MCS or/and MUS enumeration. They can be roughly divided into two classes. The first class of algorithms works with the original set of constraints and explores a lattice of all possible subsets of constraints efficiently to search for cores or/and correction sets [Liffiton and Sakallah, 2008a; Marques-Silva *et al.*, 2013; Bacchus and Katsirelos, 2015; Bacchus *et al.*, 2014; Zielke and Kaufmann, 2015; Bacchus and Katsirelos, 2016; Zhao and Liffiton, 2016; Previti *et al.*, 2018]. The second class modifies the formula at each step and extracts minimal correction sets from the modified formula [Morgado *et al.*, 2012; Marques-Silva and Planes, 2008; Liffiton and Sakallah, 2009; Alviano, 2017b]. The algorithms in this class are based on MaxSat solvers so we call this class the relaxation and strengthening framework as it iteratively transforms an unsatisfiable formula to satisfiable and vice versa. Our algorithm exploits ideas from both classes. We work within the relaxation and strengthening framework but we also keep information about the part of the lattice that we have explored so far. We extend this framework to perform core enumeration and propose a minimal correction set rotation technique that is based on ideas of model rotation used for minimization of cores [Belov and Marques-Silva, 2011].

In this work we advance state-of-the-art in MUS and MCS enumeration algorithms with the following main contributions. First, we develop an efficient MCS and MUS enumerator using the relaxation and strengthening framework. Second, we propose a new technique for MCS enumeration using a correction set rotation method. Our method allows checking minimality of a correction set efficiently without calling a SAT solver. Third, we propose a new blocking technique that maintains dependencies between relaxation variables and selector variables that enables MUS enumeration using the relaxation and strengthening framework. Finally, we preform evaluation in a prototype solver FLINT. FLINT is built on top of the PM1 core-guided MaxSAT solver [Fu and Malik, 2006]. We compare it with MARCO, the state-of-the-art method that simultaneously extracts MUSes and MCSes. We show that our approach outperforms MARCO in enumerating these objects. Moreover, our algorithm, geared toward enumerating MCSes,

significantly outperforms a variant of MARCO tuned the same way and one of the best methods for MCSes enumeration on a standard set of benchmarks [Marques-Silva *et al.*, 2013; Alviano, 2017b].

2 Background

Basic definitions. A satisfiability problem φ consists of a set of clauses $\{C_1, \ldots, C_m\}$ over a set of Boolean variables $\texttt{vars}(\varphi)$. A literal l is either a variable $x \in \texttt{vars}(\varphi)$ or its negation $\overline{x}$. A clause C is a disjunction of literals $(l_1 \vee \cdots \vee l_n)$. An assignment I of the variables $\texttt{vars}(\varphi)$ is a mapping $\texttt{vars}(\varphi) \mapsto \{0, 1\}$. A clause C is satisfied by an assignment, $I(C) = 1$, iff $I(l) = 1$ for some $l \in C$, otherwise C is falsified by I and $I(C) = 0$. A set of clauses φ is satisfied by an assignment, $I(\varphi) = 1$, iff $I(C) = 1$ for all $C \in \varphi$.

Cores and correction sets. We define two important objects of interest for unsatisfiable formulas: a minimal unsatisfiable core and a minimal correction set.

Definition 1 (MCS)**.** *A minimal correction set is a subset of clauses* MCS $\subseteq \varphi$ *such that* $\varphi \setminus$ MCS *is satisfiable and* $\forall C \in$ MCS*:* $(\varphi \setminus \text{MCS}) \cup \{C\}$ *is unsatisfiable.*

Definition 2 (MUS)**.** *A minimal unsatisfiable set is a subset of clauses* MUS $\subseteq \varphi$ *such that* MUS *is unsatisfiable and* $\forall C \in$ MUS*:* MUS $\setminus \{C\}$ *is satisfiable.*

Example 1 (Running example)**.** *Consider a formula with five clauses:* $\varphi = \{C_1, \ldots, C_5\}$*, where* $C_1 = (x_1)$*,* $C_2 = (\neg x_1 \vee x_2)$*,* $C_3 = (\neg x_2)$*,* $C_4 = (\neg x_1 \vee x_3)$*,* $C_5 = (\neg x_3)$*. We have two minimal unsatisfiable cores* $\{(C_1, C_2, C_3), (C_1, C_4, C_5)\}$*, and five minimal correction sets* $\{(C_1), (C_2, C_4), (C_2, C_5), (C_3, C_4), (C_3, C_5)\}$*.* □

During the enumeration procedure, we introduce relaxation variables, b, that are added to original clauses. For example, an original clause C_1 can be replaced with $C = (C_1 \vee b)$, where b is a relaxation variable. We denote C without relaxation variables $\texttt{orig}(C)$ and the set of relaxation variables in the clause C $\texttt{rel}(C)$. In this example, $\texttt{orig}(C_1 \vee b) = \{C_1\}$ and $\texttt{rel}(C_1 \vee b) = \{b\}$.

Each MCS is a hitting set of MUSes if we look at each MCS as a set of clauses. Dually, each MUS is a hitting set of all MCSes [Reiter, 1987].

Power sets of clauses. [Liffiton *et al.*, 2016] considered MUS/MCS enumeration as an exploration of the power set of all clauses in a formula. These subsets form a lattice by the subset relation. They introduce a notion of a `map`, which is a Boolean formula that encodes the lattice covered so far. To encode this formula, one selector variable s_i for each clause C_i is introduced, $S = \{s_1, \ldots, s_n\}$. Each time we encounter a subset of clauses $C' \subseteq \varphi$ and learn that it is a core or a correction set, we update `map` using the following *correction set blocking clauses* (1) or *core blocking clauses* (2) [1].

$$C' \text{ is a correction set} \qquad \texttt{map} = \texttt{map} \cup \{\vee_{C_i \in C'} s_i\} \quad (1)$$

$$C' \text{ is a core} \qquad \texttt{map} = \texttt{map} \cup \{\vee_{C_i \in C'} \bar{s}_i\} \quad (2)$$

[1] In [Liffiton *et al.*, 2016], the first constraint is defined for a complement of a correction set.

Example 2. *Consider the instance from Example 1. We introduce five selector variables* $(s_1, \ldots, s_5)$ *as we have five clauses. Suppose we discover that a subset of clauses* $\{C_1, C_2, C_3\}$ *is a* MUS*. We update* `map` *by adding a new clause* $(\bar{s}_1 \vee \bar{s}_2 \vee \bar{s}_3)$*. Suppose we learn that* $\{C_1\}$ *is a* MCS*. We add a clause* (s_1) *to* `map`*.* □

Note that `map` contains two types of blocking clauses. The first type contains clauses to block MCSes and the second type contains clauses to block MUSes. We call the former `mapmcs` and the latter `mapmus`, $\texttt{map} = \texttt{mapmcs} \cup \texttt{mapmus}$.

3 Enumeration Algorithm

3.1 Overview

Our algorithm is built on top of a framework proposed in [Marques-Silva and Planes, 2008; Morgado *et al.*, 2012; Alviano, 2017b] that performs a sequence of relaxations and strengthenings of the formula to obtain MCSes. At a high level, it performs a sequence of rounds until a termination condition is met. Each round consists of two phases: *Relax* and *Strengthen*. Figure 1 provides a schematic representation of this framework. In the *Relax* phase, the algorithm starts with an unsatisfiable formula and weakens it by relaxing its cores until it becomes satisfiable. The first phase is, effectively, a call to a MaxSAT solver. The resulting satisfiable formula is passed to the second phase, *Strengthen*. In the *Strengthen* phase, solutions of the satisfiable formula are enumerated and are blocked making the formula unsatisfiable again. Relaxations and strengthenings ensures that these solutions are MCSes of the formula. Then the algorithm verifies the termination condition which checks whether all MCSes have been enumerated. If so, it comes to the final stage where the remaining MUSes are enumerated using hitting set duality between MCSes and MUSes. Otherwise, the algorithm iterates with another round of strengthening and relaxation. The existing algorithms differ in the underlying MaxSAT algorithms used to relax the formula and the way they block solutions to strengthen the formula.

Here we extend this framework in several ways. First, we show that with a new solution blocking technique we can extend this framework to MUS enumeration (*RelaxCore* and *EnuMus* procedures are highlighted using italic gray in Figure 1). Second, we show that we can exploit properties of the solutions to build an efficient MCS extractor (the *MCSRotation* procedure is highlighted using italic gray in 1). Finally, we prove correctness of the algorithm.

Algorithm 1 contains the main loop of the proposed enumeration algorithm EnuMMerator. We will describe our *Relax* and *Strengthen* functions below. We start by describing global variables and structures that we use. *Relax* and *Strengthen* modify a working formula ψ^d and update global variables `projs`, `map` and `cards` that maintain necessary information about minimal cores and correction sets for the algorithm to operate. The set `projs` connects relaxation variables, b, and selector variables, s. Each time a new b^i_j variable is added to a clause C_j, we connect b^i_j with s_j as $b^i_j \Rightarrow \bar{s}_j$. These constraints ensure that if C_j is relaxed, i.e. any of its relation variables is 1, then the corresponding selector variable $s_j = 0$. The set `map` stores information about

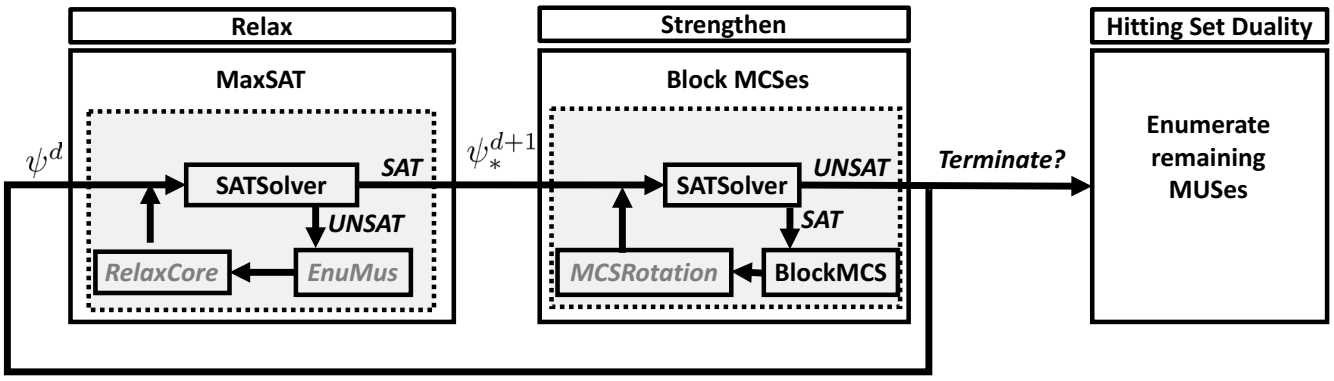


Figure 1: A schematic workflow of the relaxation and strengthening framework. We highlighted using italic gray new components of the framework that we propose in this work. *RelaxCore* contains additional constraints that connect projection and selector variables, *EnuMus* is a new MUS enumeration procedure, and *MCSRotation* is an MCS rotation based mechanism.

MCSes and MUSes found so far. Finally, `cards` stores all constraints produced by *Relax*. We will use a shortcut ρ to denote $\wedge$`cards`$\wedge$`map`$\wedge$`projs`. Using MaxSAT terminology, note that ρ cannot be relaxed, e.g., they are 'hard' clauses, in contrast to ψ^d, that are called 'soft' clauses.

Algorithm 1 EnuMMerator

Input: $\phi = \{C_1, \ldots, C_n\}$
1 **global** `cards` $= \{\}$, `map` $= \{\}$, `projs` $= \{\}$
2 $d = 0, \psi^d = \phi$
3 **while** *true* **do**
4 $\quad \psi_*^{d+1} =$ *Relax*(ψ^d)
5 $\quad \psi^{d+1} =$ *Strengthen*(ψ_*^{d+1})
6 $\quad$ **if** *Terminate*(ρ) **then**
7 $\quad\quad$ **return** HSEnumMUS(`map`)
8 $\quad d = d + 1$

Algorithm 2 *Relax*

Input: φ
Output: φ^i
1 $i = 0, \varphi^i = \varphi$
2 **while** *true* **do**
3 $\quad (issat, \kappa^i, I) =$ SolveSAT$(\varphi^i \wedge \rho)$
4 $\quad$ **if** $issat$ **then**
5 $\quad\quad$ **return** φ^i
6 $\quad$ *EnuMus*(κ^i)
7 $\quad \varphi^{i+1} =$ *RelaxCore*(φ^i, κ^i)
8 $\quad i = i + 1$

Relax

Algorithm 2 shows the *Relax* procedure, which extends PM1 [Fu and Malik, 2006]. Given an unsatisfiable formula $\varphi^0 \wedge \rho$, *Relax* performs a sequence of its relaxations. The algorithm can *only* relax clauses in φ^i. At each step, it calls a SAT solver to check whether the current formula $\varphi^i \wedge \rho$ is satisfiable (line 3). If so, it terminates. Otherwise, it finds a core $\kappa^i \subseteq \varphi^i$, relaxes the formula φ^i by relaxing clauses in κ^i by calling *RelaxCore* in line 7. On termination, the resulting formula φ^i is such that $\varphi^i \wedge \rho$ is satisfiable. Moreover, each solution of $\varphi^i \wedge \rho$ is a MCS of minimal cardinality of φ [Morgado *et al.*, 2012].

Algorithm 3 *RelaxCore*

Input: φ^i, κ^i
Output: φ^i
1 $B = \{\}$
2 **for** $C_j \in \kappa^i$ **do**
3 $\quad B = B \cup \{b_j^i\}$ where b_j^i is fresh
4 $\quad \varphi^i = (\varphi^i \setminus \{C_j\}) \cup \{(C_j \vee b_j^i)\}$
5 `cards` $=$ `cards` $\cup \{\sum_{b_j^i \in B} b_j^i = 1\}$
6 `projs` $=$ `projs` $\cup_{b_j^i \in B} \{b_j^i \Rightarrow \bar{s}_j\}$
7 **return** φ^i

Next we consider the core relaxation procedure *RelaxCore* (Algorithm 3), which relaxes soft constraints φ^i using the core κ^i. *RelaxCore* introduces a relaxation variable b_j^i for each clause $C_j \in \kappa^i$ (lines 3– 4). Then we add a cardinality constraint to ensure that exactly one of the relaxation variables can be set to *true* (line 5).

We extend the PM1 *Relax* in two ways. First, we enhance *RelaxCore* with a selector variable-based blocking technique. We implement a mapping from relaxation variables to selector variables using `projs` (line 6). Each time a new b_j^i variable is added to a clause C_j, we connect b_j^i with s_j as $b_j^i \Rightarrow \bar{s}_j$. This guarantees that if a clause C_j is disabled by I, i.e. $I(b_j^i) = 1, b_j^i \in C_j$, then the corresponding selector variable s_j is set to *false*. Second, *Relax* uses a novel MUS enumerator, *EnuMus* (line 6), to extract a subset of MUSes from κ^i. Algorithm 5 will be discussed in Section 3.2.

Example 3. *Consider the instance from Example 1. Suppose, in the first round, Relax finds a core $\kappa^0 = (C_1, C_2, C_3)$ of φ^0. It relaxes this core using RelaxCore. Namely, three relaxation variables are introduced $\{b_1^0, b_2^0, b_3^0\}$ and the corresponding clauses are modified. We get $\varphi^1 = \{C_1 \vee b_1^0, C_2 \vee b_2^0, C_3 \vee b_3^0, C_4, C_5\}$. Then we add the cardinality constraint $\{\sum_{j=1}^3 b_j^0 = 1\}$ to `cards`. Finally, we add constraints $\{b_j^0 \Rightarrow \bar{s}_j\}, j \in \{1, 2, 3\}$ to `projs` to connect relaxation variables to selector variables. The resulting formula $\varphi^1 \wedge \rho$ (recall that $\rho =$ `cards` $\wedge$ `map` $\wedge$ `projs`) is satisfiable.* □

Algorithm 4 *Strengthen*

Input: ψ

1 **while** *true* **do**
2 $(issat, I) = \text{SolveSAT}(\psi \wedge \rho)$
3 **if** $\neg issat$ **then**
4 **return**
5 $\text{MCS} = \{\cup_{I(\texttt{orig}(C_i))=0} \texttt{orig}(C_i)\}$ – original clauses false by I
6 $\texttt{map} = \texttt{map} \cup \{\vee_{C_i \in \text{MCS}} s_i\}$
7 **output** MCS
8 *MCSRotation*(ψ, MCS, I)

Strengthen

Strengthen enumerates all correction sets based on $\psi_*^{d+1} \wedge \rho$, until the strengthened ρ renders the formula unsatisfiable. To do so, the procedure enumerates solutions of $\psi_*^{d+1} \wedge \rho$. For each solution I it computes the corresponding MCS (line 5) and adds the *correction set blocking clauses* (recall (1), Section 2) to `map` (line 6). It uses a novel MCS rotation technique, *MCSRotation*, described in Section 3.3.

Example 4. *We continue with the instance from Example 3. We recall that Relax returned a satisfiable formula $\psi_*^1 = \varphi^1 \wedge \rho$. Then Strengthen finds all solutions of $\psi_*^1 \wedge \rho$. This formula has a solution where b_1^0 is assigned to 1. Hence, C_1 is relaxed in this solution. This gives a minimal correction set (C_1) and we block it by adding (s_1) to* `map`*. This makes formula unsatisfiable finishing the first round.* □

Terminate

Terminate checks whether all MCSes have been found by checking satisfiability of ρ. If so, it enumerates remaining MUSes using the hitting set duality (HSEnumMUS) (line 7).

In [Morgado *et al.*, 2012], it was shown that the relaxation and strengthening framework based on PM1 algorithm guarantees that (1) solutions of $\psi_*^{d+1} \wedge \rho$ are minimal correction sets and (2) MCSes are enumerated in the order of non-decreasing cardinality of these sets. As we use a different blocking technique for MCSes and, in addition, we block MUSes, we will need to prove these claims for EnuMMerator (see Section 3.4). Next we consider the remaining parts of EnuMMerator.

Algorithm 5 *EnuMus*

Input: κ

1 $\texttt{block} = \cup_{C_j \in \kappa} \{s_j \vee \bigvee_{b_j^i \in \texttt{rel}(C_j)} b_j^i\}$
2 **while** *true* **do**
3 $(issat, I) = \text{SolveSAT}(\rho \wedge \texttt{block})$
4 **if** $\neg issat$ **then return**
5 $g_0 = harden(\kappa \mid_I)$
6 $\kappa' = (\kappa \mid_I) \setminus g_0$
7 $\text{MUS} = g_0 \cup \text{Minimize}(g_0, \kappa')$
8 **output** MUS
9 $\texttt{map} = \texttt{map} \cup \{\vee_{C_j \in \text{MUS}} \bar{s}_j\}$

3.2 A MUS Extractor

In the core enumeration procedure, we need to extract cores of the original formula ϕ – the input formula to EnuMMerator. To do so, we will use cores κ^i that are produced by *Relax*. We apply a useful notion of a projection to transform clauses in κ^i to clauses of the original formula ϕ given an assignment I. Let I be a solution of ρ and κ, $\kappa = \kappa^i$, be a core produced at the ith iteration of *Relax*. Define a projection as follows.

$$\kappa \mid_I = \{\texttt{orig}(C_j) \mid C_j \in \kappa, \forall b_j^i \in \texttt{rel}(C_j)\ \ I(b_j^i) = 0\}.$$

Intuitively, $\kappa \mid_I$ is a subset of clauses of the original formula that are not satisfied by the solution I. [Bacchus and Narodytska, 2014] showed that given a core κ produced by PM1-based *Relax*, the projection $\kappa \mid_I$ is a core of the original formula ϕ, where I is a solution of `cards`. Let `MUS-all` be a set of all known MUSes.

First, we show that the same result holds for a solution of ρ. Second, we investigate relations between solutions of `cards` and solutions of $\rho = \texttt{cards} \wedge \texttt{map} \wedge \texttt{projs}$. We demonstrate that core blocking constraints in `map` do not guarantee that $\kappa \mid_I$ is not a superset of a known MUS in `MUS-all`. This is an undesirable property as we might keep finding known MUSes during enumeration. To fix the problem of projecting on known cores we introduce an additional set of constraints that are used *only* in *EnuMus*, `block` constraints.

Proposition 3.1. *Let I be a solution of ρ. Then $\kappa \mid_I$ is a core of the original formula ϕ.*

Proof. Suppose $\kappa \mid_I$ is not a core. Let I' be solution of $\kappa \mid_I$. We form a new solution J as follows. If $v \in vars(\kappa \mid_I)$ then $J(v) = I'(v)$, otherwise $J(v) = I(v)$. Next, we note that $vars(\kappa \mid_I) \cap vars(\rho) = \{\}$. Hence, we do not change the assignment of projection and selector variables in J compared to I. Note that clauses in $\kappa \setminus (\kappa \mid_I)$ are satisfied using the corresponding projection variables b_j^i that are set to 1 in I, hence, they are also set to 1 in J. $\kappa \mid_I$ is satisfied as $vars(\kappa \mid_I)$ are assigned as in I'. Finally, ρ is satisfied as we did not change assignment of variables in ρ in J compared to I. Hence, J is a solution of $\kappa \wedge \rho$, which leads to a contradiction. □

The next example shows that `map` and `projs` are not sufficient to eliminate all known cores.

Example 5. *We use the instance from Example 4. Consider the second round and assume that EnuMus was called in the first round and recorded the core it found in* `map`*. At this point,* `cards` *and* `projs` *are as in Example 4 and* `map` $= \{(s_1), (\bar{s}_1 \vee \bar{s}_2 \vee \bar{s}_3)\}$*. Suppose Relax finds a core $\kappa^0 = (C_1 \vee b_1^0, C_4, C_5)$. EnuMus is called. Consider two solutions of ρ: $I_1 = (b_1^0 = 0, b_2^0 = 1, b_3^0 = 0, s_1 = 1, s_2 = \ldots = s_5 = 0)$, and $I_2 = (b_1^0 = 0, b_2^0 = 0, b_3^0 = 1, s_1 = 1, s_2 = \ldots = s_5 = 0)$. These solutions are identical except for swapped values of b_2^0 and b_3^0. Suppose we find I_1 first and project it to get a core $\kappa^0 \mid_{I_1} = \{C_1, C_4, C_5\}$. We can block this core by adding the clause $(\bar{s}_1 \vee \bar{s}_4 \vee \bar{s}_5)$ to* `map`*. However, this clause does not eliminate I_2 that is projected to the same core: $\kappa^0 \mid_{I_2} = \{C_1, C_4, C_5\}$.*

The main issue here is that C_4 and C_5 are always selected in a projection as they do not contain projection variables. However, there are no constraints to enforce that if a clause is in a projection then the corresponding selector variable is set to 1*. So, we cannot enforce that for any solution I if $C_5/C_4 \in \kappa^0 \mid_I$ then s_5/s_4 must be 1 in this example.* □

From Example 5 it follows.

Proposition 3.2. *There exists a solution I of ρ such that $m \subseteq \kappa \mid_I$, $m \in$ `MUS-all`.*

Next we describe our MUS enumerator, *EnuMus* (Algorithm 5). *EnuMus* takes a core κ as an input. By Proposition 3.1, we just need to enumerate solutions of ρ to extract

cores of the original formula ϕ. *EnuMus* performs such enumeration. However, by Proposition 3.2 the enumeration procedure will produce a lot of solutions that correspond to MUSes that we have encountered before. We introduce additional constraints, `block`, to make sure that if a clause is in a projection, i.e. $C_j \in \kappa\mid_I$, then the corresponding selector variable $s_j = 1$ (line 1, *EnuMus*). $\texttt{block} = \cup_{C_j \in \kappa}\{s_j \vee \bigvee_{b_j^i \in \texttt{rel}(C_j)} b_j^i\}$. Hence, if $\kappa\mid_I$ is a superset of a known MUS m then the core blocking clause $(\vee_{C_j \in m} \bar{s}_j)$ is falsified. The `block` constraints resolve the issue pointed out in Example 5.

Proposition 3.3. *If I is a solution of ρ then $\forall m \in$ `MUS-all` we have that $m \nsubseteq \kappa\mid_I$.*

Proof. Let $\pi = \kappa\mid_I$. The proof follows from two observations. First, we know that $I(b_j^i) = 0, \forall b_j^i \in \texttt{rel}(C_j), C_j \in \pi$. Hence, `block` clauses enforce that $s_j = 1$, which implies that $(\vee_{C_j \in \pi} \bar{s}_j)$ is falsified by I and cannot be contained in `map`. As `map` blocks all MUSes that have been found so far, we know that π is not contained in known MUSes. □

Algorithm 5 shows pseudocode. *EnuMus* takes a core κ as an input. Then it enumerates solutions of $\rho \wedge \texttt{block}$. By Proposition 3.3, if there exists a solution I of $\rho \wedge \texttt{block}$ then $\kappa\mid_I$ contains a new MUS of the original formula ϕ. Before minimizing $\kappa\mid_I$, we perform simple preprocessing based on the duality between cores and correction sets. The *harden* procedure (line 6) returns g_0 as the clauses that must be in the minimal core: $C \in (\kappa\mid_I)$ s.t. $\text{MCS} \cap (\kappa\mid_I) = \{C\}$ for some MCS that we found so far. The same optimization was used in [Liffiton *et al.*, 2016; Zielke and Kaufmann, 2015]. Finally, we minimize κ' with respect to g_0 as hard constraints and block it.

Example 6. *We continue with the instance from Example 4. Consider the second round and assume that EnuMus was called in the first round and recorded the core it found in* `map`*. At this point,* `cards` *and* `projs` *are as in Example 4 and* $\texttt{map} = \{(s_1), (\bar{s}_1 \vee \bar{s}_2 \vee \bar{s}_3)\}$*. Suppose Relax finds a core* $\kappa^0 = (C_1 \vee b_1^0, C_4, C_5)$*. EnuMus is called. We form* `block` *constraints:* $\texttt{block} = \{(s_4), (s_5)\}$ *as there are no projection variables in C_4/C_5. The first solution of* $(\rho \wedge \texttt{block})$ *is* $I = (b_1^0 = 0, b_2^0 = 1, b_3^0 = 0, s_1 = 1, s_2 = s_3 = 0, s_4 = s_5 = 1)$*. The corresponding projection* $\kappa^0\mid_I = \{C_1, C_4, C_5\}$*. We check that* $\kappa^0\mid_I$ *is minimal and block this core by adding the clause* $(\bar{s}_1 \vee \bar{s}_4 \vee \bar{s}_5)$ *to* `map`*. Note that in contrast with Example 5, s_4 and s_5 must take values 1 due to* `block` *constraints. This allows us to eliminate solutions that are mapped to known* MUS*es, like I_2.* □

3.3 Enhanced MCS Extractor

In this section we consider a new approach to speed up the MCS enumeration procedure. *Strengthen* takes ψ as the input and finds all solutions. However, the size of the formula grows at each iteration and finding new solutions becomes expensive. Here we propose a novel approach to speed up MCS enumeration which is very efficient in practice.

We will show in Section 3.4 that EnuMMerator finds MCSes of non-decreasing cardinalities of the original formula ϕ. Let $\texttt{MCS-all}_{<k}$ be a set of *all* MCSes of ϕ of size less than k.

Proposition 3.4. *Let m be a correction set of size k. The correction set m is minimal iff* $\forall \text{MCS} \in \texttt{MCS-all}_{<k}$ $\text{MCS} \not\subset m$.

Proof. As m is not a superset of any MCS of size smaller than k, then it is minimal by definition. □

Algorithm 6 *MCSRotation*

Input: ψ, MCS, I

```
1  ψ = orig(ψ)
2  MCS-all = getmcses(map)
3  for C ∈ MCS do
4      for l_i ∈ C do
5          I(l_i) = 1
6          m' = ∪_{I(C)=0, C∈ψ} {C}
7          if |m'| = |MCS| ∧ ∀m ∈ MCS-all m ⊄ m' then
8              map = map ∪ {∨_{C_i∈m'} s_i}
9              output m'
10             MCSRotation(ψ, m', I)
11         I(l_i) = 0
```

Algorithm 6 shows pseudocode. We first obtain the original formula by removing all relaxation variables from it (line 1) and get all MCSes from the `map` using $getmcses(\texttt{map})$ (line 2). In practice, we keep all MCSes in a data structure so we do not need to extract MCSes from `map` each time. Our rotation is inspired by to the model rotation techniques from [Belov and Marques-Silva, 2011] which is an efficient technique for core minimization. We start from an assignment I and the corresponding MCS. We pick a clause C from MCS and a literal $l_i \in C$. As C is in the MCS, then $I(C) = 0$. Then we satisfy a literal l_i by flipping $I(l_i) = 0$ to $I(l_i) = 1$ (line 5). This makes C satisfiable but other clauses might become violated. We compute a correction set m' that corresponds to a new assignment (line 6). If the size of m' is equal to the size of MCS and $\forall m \in \texttt{MCS-all}$, m is *not* subset of m' then, by Proposition 3.4, we know that m' is a MCS (line 7). Then we recursively call *MCSRotation* (line 10).

Example 7. *We continue with the instance from Example 4. We focus on Strengthen at the second round. Recall that Relax returned a satisfiable formula $\psi_*^2 = \varphi^1 \wedge \rho$. This formula has four solutions that correspond to $MCSes$: (C_2, C_4), (C_2, C_5), (C_3, C_4) and (C_3, C_5). So the Strengthen algorithm has to make 4 SAT calls to find them. With MCSRotation, Strengthen can reduce the number of calls to the SAT solver. Suppose Strengthen finds the first solution I and passes it to MCSRotation. Suppose $I = (x_1 = 1, x_2 = 0, x_3 = 0)$ and the corresponding* MCS *is $\{C_2, C_4\}$ (see Example 1 for the definition of clauses). At this point* $\texttt{MCS-all}_{\leq 2} = \{(C_1)\}$*. Suppose we flip $x_2 = 0$ to $x_2 = 1$. The new assignment violates $m' = \{C_3, C_4\}$. As $|m'| = 2$ we check that none of* MCS*es in* $\texttt{MCS-all}_{\leq 2}$ *is a subset of m'. Hence, we know that m' is a new* MCS *without calling a SAT solver.* □

3.4 Correctness

In this section, we sketch a proof of correctness.

Theorem 3.5. *EnuMMerator finds all* MCS*es and* MUS*es when it terminates.*

Proof. (Sketch) We first prove correctness of a simpler version of EnuMMerator where *EnuMus* is omitted. We recall

that map contains two types of blocking clauses: map = mapmcs ∪ mapmus. If we exclude *EnuMus*, we have that map = mapmcs.

A simplified EnuMMerator enumerates all MCSes and uses hitting set duality to enumerate all MUSes. Hence, we can reuse the proof of correctness of the the relaxation and strengthening framework for finding MCSes as in [Morgado *et al.*, 2012]. We only need to prove correctness of our MCS blocking procedure and the termination condition.

The main part to prove is that we never block a MCS that we have not produced. Consider the ith iteration. We prove this by contradiction. Suppose, there exists a MCS m that has never been encountered during the execution of EnuMMerator. Suppose, it was eliminated after adding a projection constraint to projs at the ith iteration. So, we have that $\psi^{d+1} \wedge \rho$ has a solution I that correspond to m and $\psi^{d+1} \wedge \rho \wedge (b^i_j \Rightarrow \bar{s}_j)$ eliminates this solution. We form a new solution $I' = I$ and set $I'(s_j) = 1$. Note that I' must satisfy $\psi^{d+1} \wedge \rho$ as setting s_j to 1 will not violate any constraint. It also satisfies $b^i_j \Rightarrow \bar{s}_j$ as $I(s_j) = 1$. On the other hand, I' corresponds to the same MCS m, as the assignment of projection variables does not change. Therefore, $b^i_j \Rightarrow \bar{s}_j$ cannot eliminate m. Hence, it should be eliminated by the *correction set blocking clauses* (1) at the ith iteration on seeing MCS m'. Note that the clause $\vee_{C_i \in m'} s_i$ we add to mapmcs only eliminates m' and its supersets. Hence, either $m = m'$ or it is not minimal. This leads to a contradiction.

Next we come back to EnuMMerator with *EnuMus*. We prove that *EnuMus* does not change the solution space of $\psi^d \wedge$ cards $\wedge$ projs $\wedge$ mapmcs. Namely, we prove that any solution $\psi^d \wedge$ cards $\wedge$ projs $\wedge$ mapmcs can be extended to a solution of mapmus. Let I be a solution of $\psi^d \wedge$ cards $\wedge$ projs $\wedge$ mapmcs. Consider an MCS of the original formula ϕ that corresponds to this solution I. First, we note that there exists $b^i_j \in$ rel(C_j), $C_j \in$ MCS such that $I(b^i_j) = 1$. Then, $I(s_j)$ must be 0 for all j such that $C_j \in$ MCS due to projs clauses. The main observation is that any MCS hits all MUSes. Therefore, for each clause $(\bar{s}_{j_1} \vee \ldots \bar{s}_{j_n}) \in$ mapmus, at least one of the literals is set to 0. Hence, all clauses in mapmus are satisfied. □

4 Related Work

We briefly overview two types of closely related work. The first line of related research proposed a relaxation and strengthening framework for MCS enumeration [Marques-Silva and Planes, 2008; Morgado *et al.*, 2012; Alviano, 2017b]. The *Relax* procedure was performed using MSU3, PM1 and OLL MaxSat algorithms respectively. We use the same framework but propose a different blocking technique that allows us to enumerate cores as well as correction sets. Moreover, we propose a procedure to perform core extraction and MCS rotation to speed up MCS enumeration.

The second line of work develops methods that work with the original formula [Liffiton and Sakallah, 2008a; Bacchus *et al.*, 2014; Zielke and Kaufmann, 2015; Bacchus and Katsirelos, 2016; Previti *et al.*, 2017]. Our approach is based on ideas from the MARCO algorithm that explores a lattice of all possible subsets of clauses. However, instead of explicit exploration of points of the lattice, we extract *sets* of points at each iteration. These sets come from solutions of satisfiability problems that we either build incrementally (for MCSes) or form dynamically (for MUSes) during search.

A number of algorithms were proposed that focus on MCS or MUS enumeration. [Previti *et al.*, 2017] proposed keeping track of unsatisfiable cores using a new caching mechanism, and proposed an efficient way to check if a given sub-formula contains a known core. It is a matter of future research to explore if we can use a caching mechanism within the relaxation and strengthening framework. It will likely be different from [Previti *et al.*, 2017], as we do not grow a maximal satisfying assignment to obtain MCSes. [Bacchus *et al.*, 2014] proposed a relaxation search that uses a SAT solver to efficiently find sets of optional clauses to remove. [Zielke and Kaufmann, 2015] proposed to consider the *block property* of a set of clauses. These clauses either occur together in a MUS or do not occur at all. The discovered sets with this property can be used to speed up MARCO. [Bacchus and Katsirelos, 2016] show that MARCO can be significantly improved by incrementally enumerating cores and proposed other improvements. They demonstrated that any MARCO-like algorithm can leverage incremental core enumeration if it is deeply integrated with MCSMUS. The key idea behind the incremental part is that MCSMUS can take advantage of enumerated MCSes to produce MUSes incrementally. Such deep integration is one possible avenue for improving our algorithm.

5 Experimental Evaluation

We work with a standard set of 295 benchmarks from the MUS track of the 2011 SAT competition [Jarvisalo *et al.*, 2011]. We ran our experiments on Intel(R) Xeon(R) 3.50GHz. We compare our algorithm, called FLINT, with MARCO, the state-of-the-art method that simultaneously extracts MUSes and MCSes. We build FLINT on top of PM1 from [Janota, 2013]. For fair comparison, we call the MUSer algorithm [Belov and Marques-Silva, 2012] as a standalone procedure for core minimization the same way as MARCO. As our algorithm can also be geared toward correction sets enumeration, we consider a version that only enumerates correction sets (FLINT-MCS) and we compare it with MARCO-MCS (called MARCO-MSS in [Liffiton *et al.*, 2016]), which is a version of MARCO biased toward MCS enumeration. Finally, we compare with several algorithms that focus on enumeration of only MUSes/MCSes. We use timeout of 3600 sec in all runs.

We discuss implementation details. First, we consider how we preform the forward subsumption check in line 7 of Algorithm 6. Recall that it suffices to perform only forward subsumption as none of the existing MCSes is a superset of a candidate correction set m. In the benchmark set, the size of correction sets that we find is on average 5, and generally less than 10. So instead of using a general discrimination tree index or index based on feature vectors [Schulz, 2013], it suffices to index each MCS of length n into a table (trie or hash-table) that contains all size n sets. Then $\binom{|m|}{n}$ lookups are performed for every $n = 1, \ldots, |m|$. In our experiments we used a trie to store MCSes [Tessil, 2017]. Second, we discuss the MUS extractor. We found that κ' (Algorithm 5,

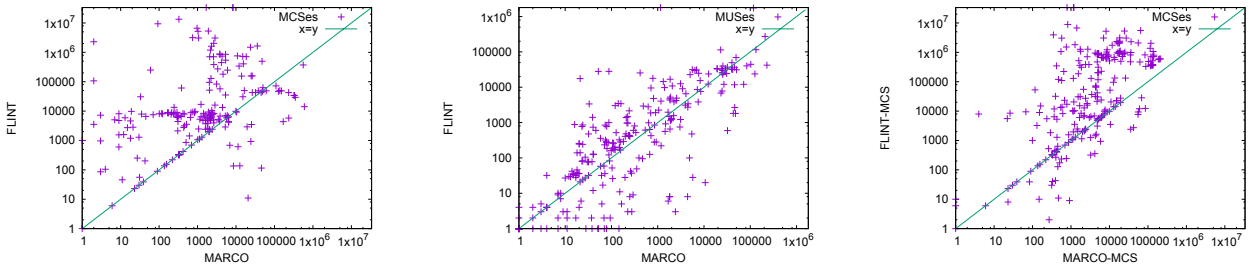


Figure 2: Comparison of FLINT and MARCO algorithms: (a) the number of MCSes produced by FLINT and MARCO; (b) the number of MUSes produced by FLINT and MARCO; (c) the number of MCSes produced by FLINT-MCS and MARCO-MCS.

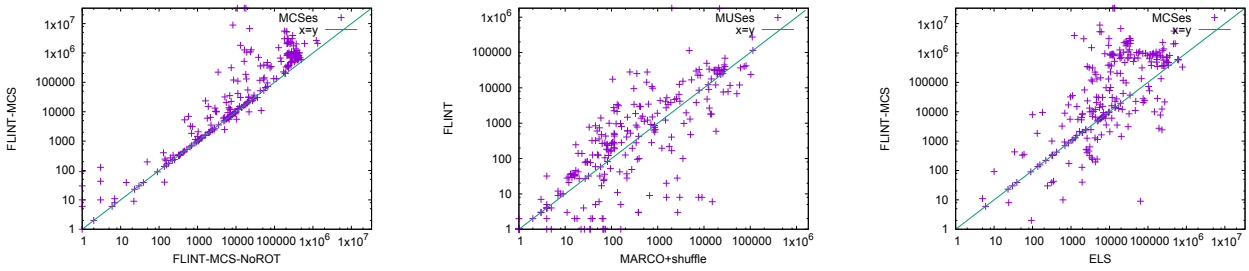


Figure 3: (a) Comparison of FLINT-MCS and FLINT-MCS-noROT modification algorithms that does not use *MCSRotation* in terms of the number of MCSes; (b) Comparison of FLINT and MARCO+shuffle heuristics in terms of the number of MUSes; (c) Comparison of FLINT-MCS and ELS algorithms in terms of the number of MCSes.

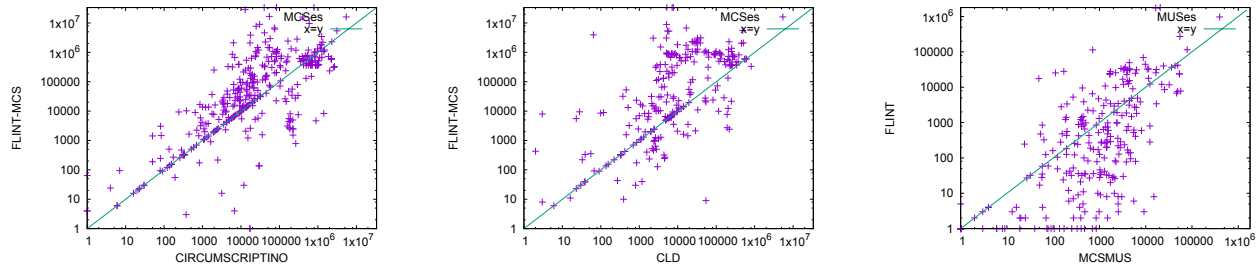
line 6) often contains multiple minimal cores. This might be related to the fact that the input core κ is not minimal as it is very expensive to minimize it. However, MUSer only finds one core per call. We employ a simple heuristic where we randomly shuffle clauses in κ' and give it to MUSer again as it often produces different cores. We filter out repeated cores and terminate shuffling if we start encountering many known MUSes (five in a row in our implementation). Since this algorithm relies on randomization, we run it five times and show the average result.

5.1 Comparison with MARCO

We compare our algorithm with MARCO. First we compare the algorithms by number of MUSes and MCSes generated within 3600s. Figures (2a)–(2b) show the corresponding log scale scatter plots. In all plots, the more points are above the $x = y$ line the better FLINT/FLINT-MCS performs compared to the competing algorithm. As can be seen from these plots, FLINT outperforms MARCO in the enumeration of both objects (more points above the $x = y$ line indicates that FLINT outperforms MARCO). FLINT finds more MCSes in 166 instances and more MUSes in 143 instances compared to MARCO. On the other hand, MARCO finds more MCS in 54 instances and more MUSes in 97 instances compared to FLINT. The algorithms ties on the remaining instances. We also compared 'biased' to the MCSes versions of FLINT and MARCO (see Figures (2c)). Again, we observe that FLINT-MCS performs much better compared to MARCO-MCS: FLINT-MCS finds more MCS in 202 instances and fewer MCSes in 55 instances compared to MARCO-MCS. We also investigate the contribution of the *MCSRotation* procedure to the performance of FLINT-MCS (Figure 3a). To do so, we run FLINT-MCS with and without MCS rotation. *MCSRotation* helps in most instances (187 instances) and hurts in only a small fraction of instances (22 instances). Finally, we compare with MARCO enhanced with the same shuffling heuristics for MUS extraction as used in our algorithm. To do so, each time we get a seed which results to unsatisfiable subformula, the shuffle clauses in this seed and pass it to MUSer. If we encounter five known MUSes in a row we terminate stop shuffling. Figure 3b shows the results. We found that the shuffling heuristics slightly degrades performance of MARCO. For example, FLINT finds more MUSes in 153 instances compared to MARCO+shuffling variants.

5.2 Comparison with Other Algorithms

We compare FLINT-MCS with state-of-the-art algorithms that focus on enumeration of only one type of objects: MCSes or MUSes. First, we consider several state-of-the-art algorithms for MCS enumeration which are ELS and CLD for propositional formulas and CIRCUMSCRIPTINO for propositional theories. To compare with ELS and CDL, we use the same standard set of benchmarks. FLINT-MCS finds more MCSes in 167 instances, fewer in 76 instances compared to ELS and tie on the remaining instances (Figure 3c). FLINT-MCS finds more MCSes in 173 instances, fewer in 65 instances com-



(a) (b) (c)

Figure 4: (a) Comparison of FLINT-MCS and CIRCUMSCRIPTINO in terms of the number of MCSes; (b) Comparison of FLINT-MCS and CLD in terms of the number of MCSes; (c) Comparison of FLINT-MCS and MCSMUS in terms of the number of MUSes.

pared to CLD and tie on the remaining instances (Figure 4b). In case of CIRCUMSCRIPTINO, we used a different dataset as it does not accept the DIMACS CNF input format; hence we compared on the benchmark set provided by the authors that contains instances encoded as a propositional theory and as DIMACS CNF [Alviano, 2017a]. The total number of instances in 395 in this dataset. FLINT-MCS finds more MCSes in 210 instance and fewer MCSes in 99 instances compared to CIRCUMSCRIPTINO (Figure 4a).

Overall, these results show that the MCS-biased version of FLINT performs very well compared to algorithms that only focus on MCSes. We also compare FLINT with the MCSMUS algorithm, that focuses on MUS enumeration [Bacchus and Katsirelos, 2015; Bacchus and Katsirelos, 2016]. As the non-incremental version of this algorithm is less efficient than MARCO [Bacchus and Katsirelos, 2015], we do not consider it here. Instead, we consider a version of MCSMUS that computes MUSes incrementally. FLINT finds more MUSes in 85 and fewer MUSes in 171 instances compared to MCSMUS (Figure 4c). This shows that FLINT is competitive with an incremental MCSMUS. It also shows that our algorithm can potentially be improved by deep integration of incremental core enumeration [Bacchus and Katsirelos, 2016].

5.3 Discussion of Results

We discuss how our algorithm can be extended based on several observations from our experiments.

As the solver is based on the relaxation and strengthening framework, the choice of the relaxation technique that is borrowed from MaxSat has strong influence on the performance. Currently, we use the PM1 algorithm, which is easier to implement. However, we observed that in the benchmark set, there are 32 instances where we were not able to complete the first relaxation step using PM1. A more efficient relaxation procedure might help on these instances. The PM1 algorithm can be replaced with other cardinality-based relaxation techniques, e.g. MaxRes, OLL, ONE, etc, [Andres *et al.*, 2012; Narodytska and Bacchus, 2014; Alviano *et al.*, 2015]. Common to these algorithms is the requirement to keep track of all introduced relaxation constraints and make sure that these constraints satisfy Proposition 3.1.

Core minimization is an important and, often, the most expensive part. Several improvements over FLINT are possible. The feedback loop between cores and correction sets can be improved. While the *harden* procedure [Previti *et al.*, 2017; Zielke and Kaufmann, 2015] is a simplistic preprocessing step, yet we observed it can identify up to 30% of clauses in κ as necessarily members of the cores. To this end, it is interesting to build a solver with deep integration of MCSMUS as described in [Bacchus and Katsirelos, 2016]. Finally, it should be possible to take advantage of κ' containing multiple cores produced by *Relax*. We observed that a lot of cores produced from κ' share a large number of clauses. This can be exploited in core minimization.

6 Conclusion

In this work, we proposed an algorithm to enumerate minimal correction sets and cores simultaneously. We combined the relaxation and strengthening framework with lattice exploration methods. We propose several new techniques that allow efficient core and correction set enumeration. Our PM1 based prototype outperforms MARCO in enumeration of MUSes and MCSes on a standard set of benchmarks.

References

[Alviano *et al.*, 2015] Mario Alviano, Carmine Dodaro, and Francesco Ricca. A maxsat algorithm using cardinality constraints of bounded size. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015*, pages 2677–2683. AAAI Press, 2015.

[Alviano, 2017a] Mario Alviano. Circumscriptino. https://alviano.com/software/circumscriptino/, 2017.

[Alviano, 2017b] Mario Alviano. Model enumeration in propositional circumscription via unsatisfiable core analysis. *TPLP*, 17(5-6):708–725, 2017.

[Andres *et al.*, 2012] Benjamin Andres, Benjamin Kaufmann, Oliver Matheis, and Torsten Schaub. Unsatisfiability-based optimization in clasp. In *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012*, volume 17, pages 211–221, 2012.

[Bacchus and Katsirelos, 2015] Fahiem Bacchus and George Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *Computer Aided Verification - 27th International Conference, CAV 2015, July 18-24, 2015*, volume 9207, pages 70–86, 2015.

[Bacchus and Katsirelos, 2016] Fahiem Bacchus and George Katsirelos. Finding a collection of muses incrementally. In *Integration of AI and OR Techniques in Constraint Programming - 13th International Conference, CPAIOR 2016*, volume 9676, pages 35–44, 2016.

[Bacchus and Narodytska, 2014] Fahiem Bacchus and Nina Narodytska. Cores in core based maxsat algorithms: An analysis. In *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014*, volume 8561, pages 7–15, 2014.

[Bacchus *et al.*, 2014] Fahiem Bacchus, Jessica Davies, Maria Tsimpoukelli, and George Katsirelos. Relaxation search: A simple way of managing optional clauses. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, 2014*, pages 835–841, 2014.

[Belov and Marques-Silva, 2011] Anton Belov and Joao Marques-Silva. Accelerating MUS extraction with recursive model rotation. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, 2011*, pages 37–40. FMCAD Inc., 2011.

[Belov and Marques-Silva, 2012] Anton Belov and Joao Marques-Silva. Muser2: An efficient MUS extractor. *JSAT*, 8(3/4):123–128, 2012.

[Belov *et al.*, 2012] Anton Belov, Ines Lynce, and Joao Marques-Silva. Towards efficient MUS extraction. *AI Commun.*, 25(2):97–116, 2012.

[Fu and Malik, 2006] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, 2006*, volume 4121, pages 252–265, 2006.

[Janota, 2013] Mikoas Janota. The MiFuMaX solver. http://sat.inesc-id.pt/ mikolas/sw/mifumax/, 2013.

[Jarvisalo *et al.*, 2011] Matti Jarvisalo, Daniel Le Berre, Joao Marques-Silva, and Olivier Roussel. The MUS track of the 2011 SAT competition. http://www.satcompetition.org/2011/, 2011.

[Junker, 2004] Ulrich Junker. QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, pages 167–172, 2004.

[Liffiton and Sakallah, 2008a] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008.

[Liffiton and Sakallah, 2008b] Mark H. Liffiton and Karem A. Sakallah. Reveal: A formal verification tool for verilog designs. In *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008*, volume 5330, pages 343–352, 2008.

[Liffiton and Sakallah, 2009] Mark H. Liffiton and Karem A. Sakallah. Generalizing core-guided max-sat. In *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009*, volume 5584, pages 481–494. Springer, 2009.

[Liffiton *et al.*, 2016] Mark H. Liffiton, Alessandro Previti, Ammar Malik, and Joao Marques-Silva. Fast, flexible MUS enumeration. *Constraints*, 21(2):223–250, 2016.

[Marques-Silva and Planes, 2008] Joao Marques-Silva and Jordi Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In *Design, Automation and Test in Europe, DATE 2008, Munich, Germany, March 10-14, 2008*, pages 408–413. ACM, 2008.

[Marques-Silva *et al.*, 2013] Joao Marques-Silva, Federico Heras, Mikolas Janota, Alessandro Previti, and Anton Belov. On computing minimal correction subsets. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence 2013*, pages 615–622, 2013.

[Morgado *et al.*, 2012] Antonio Morgado, Mark H. Liffiton, and Joao Marques-Silva. Maxsat-based MCS enumeration. In *HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers*, volume 7857, pages 86–101. Springer, 2012.

[Narodytska and Bacchus, 2014] Nina Narodytska and Fahiem Bacchus. Maximum satisfiability using core-guided maxsat resolution. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, 2014.*, pages 2717–2723. AAAI Press, 2014.

[Previti *et al.*, 2017] Alessandro Previti, Carlos Mencia, Matti Jarvisalo, and Joao Marques-Silva. Improving MCS enumeration via caching. In *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, 2017*, volume 10491, pages 184–194, 2017.

[Previti *et al.*, 2018] Alessandro Previti, Carlos Mencia, Matti Jarvisalo, and Joao Marques-Silva. Premise set caching for enumerating minimal correction subsets. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, 2018*, 2018.

[Reiter, 1987] Raymond Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987.

[Schulz, 2013] Stephan Schulz. Simple and efficient clause subsumption with feature vector indexing. In *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*, volume 7788, pages 45–67, 2013.

[Tessil, 2017] Thibaut Tessil. A C++ implementation of a fast and memory efficient HAT-trie. https://github.com/Tessil/hat-trie/, 2017.

[Zhao and Liffiton, 2016] Wenting Zhao and Mark H. Liffiton. Parallelizing partial MUS enumeration. In *28th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2016*, pages 464–471. IEEE Computer Society, 2016.

[Zielke and Kaufmann, 2015] Christian Zielke and Michael Kaufmann. A new approach to partial MUS enumeration. In *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, 2015*, volume 9340, pages 387–404, 2015.