EPiC Series in Computing
Volume 73, 2020, Pages 120–137
LPAR23. LPAR-23: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning


# Rotation Based MSS/MCS Enumeration*

Jaroslav Bendík and Ivana Černá

Faculty of Informatics, Masaryk University, Brno, Czech Republic
{xbendik,cerna}@fi.muni.cz

**Abstract**

Given an unsatisfiable Boolean Formula $F$ in CNF, i.e., a set of clauses, one is often interested in identifying Maximal Satisfiable Subsets (MSSes) of $F$ or, equivalently, the complements of MSSes called Minimal Correction Subsets (MCSes). Since MSSes (MCSes) find applications in many domains, e.g. diagnosis, ontologies debugging, or axiom pinpointing, several MSS enumeration algorithms have been proposed. Unfortunately, finding even a single MSS is often very hard since it naturally subsumes repeatedly solving the satisfiability problem. Moreover, there can be up to exponentially many MSSes, thus their complete enumeration is often practically intractable. Therefore, the algorithms tend to identify as many MSSes as possible within a given time limit. In this work, we present a novel MSS enumeration algorithm called RIME. Compared to existing algorithms, RIME is much more frugal in the number of performed satisfiability checks which we witness via an experimental comparison. Moreover, RIME is several times faster than existing tools.

## 1 Introduction

In many areas of computer science, we are given a Boolean formula $F$ in a conjunctive normal form (CNF), i.e., a set $F = \{f_1, \ldots, f_n\}$ of Boolean clauses, with the goal to decide the satisfiability of $F$. If $F$ is found to be unsatisfiable, one is often interested in identifying a maximal satisfiable subset (MSS) of clauses of $F$. Equivalently, one can identify the complement of a MSS called minimal correction subset (MCS) of $F$, i.e., a minimal set of clauses that need to be removed from $F$ to make it satisfiable.

The identification of MSSes (MCSes) finds many practical applications including, e.g., computation of minimal models of CNF formulas and model based diagnosis [12], ontology debugging or axiom pinpointing [1]. Another important application of MSSes emerges for example in the context of the maximum satisfiability (MaxSAT) problem. In particular, the MSSes with the largest cardinality are the exact solutions of MaxSAT, and MSSes with smaller cardinalities can be used to at least efficiently approximate the solution of MaxSAT [27].

In general, the more MSSes (MCSes) are identified, the better insight into the unsatisfiability of $F$ is obtained. However, the complete MSS (MCS) enumeration is often practically

*This research was supported by ERDF "CyberSecurity, CyberCrime and Critical Information Infrastructures Center of Excellence" (No. CZ.02.1.01/0.0/0.0/16_019/0000822).

intractable, since there can be up to exponentially many MSSes w.r.t. the number of clauses in $F$. Therefore, there have been proposed several algorithms that enumerate MSSes *online*, i.e., one by one, and can be terminated at any time (see, e.g., [5, 37, 26, 27, 31]).

Many of the online algorithms can be classified as *seed-grow* algorithms. A seed-grow algorithm gradually explores individual subsets of $F$; explored subsets are those whose satisfiability is already determined by the algorithm and unexplored are the other ones. Each single MSS is found in two steps. First, the algorithm identifies a *seed* $S$, i.e., a satisfiable unexplored subset. Then, the seed $S$ is *grown* into a MSS $S_{mss}$ of $F$ such that $S \subseteq S_{mss}$. The exact ways of finding and growing the seeds differ for individual seed-grow algorithms. In general, all existing seed-grow algorithms find a seed by repeatedly picking and checking an unexplored subset for satisfiability (via a SAT solver) until a satisfiable subset is found. The growing is typically based on checking several supersets of the seed for satisfiability until a MSS is found. The efficiency of individual algorithms, i.e., the speed of the MSS enumeration, usually depends on the total number of satisfiability checks performed during finding and growing the seeds. The fewer satisfiability checks are performed to find each single MSS, the faster is the enumeration.

In this paper, we propose a novel seed-grow MSS enumeration algorithm called RIME. To find seeds, RIME employs two novel techniques. One technique works on the same principle as the existing seed-grow algorithms do: we repeatedly check unexplored subsets for satisfiability until we find a seed. The novelty is in the selection of unexplored subsets; we use the intersection of already explored MSSes of $F$ to guide the search among unexplored subsets and we identify seeds that are very close to MSSes (and thus can be easily grown). The other technique is fundamentally different: we exploit the MSSes that are already explored to cheaply *deduce* that some unexplored subsets are satisfiable, i.e., seeds, without using a SAT solver. Moreover, we propose a novel growing procedure that greatly benefits from the information about already explored subsets of $F$. In particular, we introduce a concept of *conflict extension* that allows us to significantly enlarge seeds during growing. We experimentally compare RIME with 3 other contemporary MSS enumeration algorithms on a large collection of benchmarks. RIME significantly outperforms its competitors on a majority of the benchmarks in the number of identified MSSes within a given time limit. Remarkably, RIME needs to perform much fewer calls to a SAT solver to identify individual MSSes than its competitors do. On average, RIME performs just 1.18 satisfiability checks per MSS, and in case of many benchmarks, the number of checks per MSS is even smaller than 1.

## 2 Preliminaries

A Boolean formula $F = \{f_1, f_2, \ldots, f_n\}$ in a conjunctive normal form (CNF) is a set of Boolean clauses over a set of Boolean variables $\mathit{Vars}(F)$. A Boolean clause is a set $\{l_1, l_2, \ldots, l_k\}$ of literals. A literal is either a variable $x \in \mathit{Vars}(F)$ or its negation $\neg x$. A truth assignment $\pi$ to the variables $\mathit{Vars}(F)$ is a mapping $\mathit{Vars}(F) \to \{\top, \bot\}$. A clause $f \in F$ is satisfied by an assignment $\pi$ iff $\pi(l) = \top$ for some $l \in f$ or $\pi(k) = \bot$ for some $\neg k \in f$. The formula $F$ is satisfied by $\pi$ iff $\pi$ satisfies every $f \in F$; in such a case $\pi$ is called a *model* of $F$. Finally, $F$ is *satisfiable* if it has a model; otherwise $F$ is *unsatisfiable*.

Throughout the whole text, we write just *formula* instead of the full term *Boolean formula in CNF*. Similarly, we write just *clauses* and *variables*. Moreover, in the whole text, we use $F$ to denote an unsatisfiable formula that we want to analyze. We use capital letters, e.g., $C, N, K$ to denote subsets of $F$, small letters, e.g., $f, f_i, g$, to denote clauses, and small letters, e.g., $x, x', y$, to denote variables. Given a set $X$, we write $\mathcal{P}(X)$ to denote the power-set of $X$, and $|X|$ to denote the cardinality of $X$.

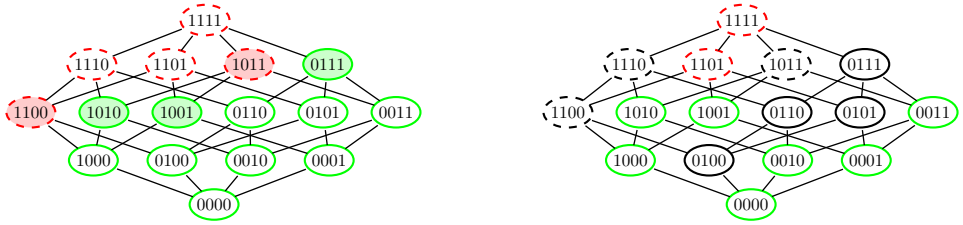



Figure 1: Fig. 1a: illustration of the power set of the set $F$ of clauses from Example 1. We encode individual subsets of $F$ as bit-vectors; for example, the subset $\{f_1, f_3, f_4\}$ is written as 1011. The subsets with a dashed border are the unsatisfiable subsets, and the others are satisfiable subsets. The MUSes and MSSes are filled with a background color. Fig. 1b: illustration of the explored and unexplored subsets from Example 2.

## 2.1 Maximal Satisfiability and Minimal Unsatisfiability

**Definition 1** (MSS)**.** *A set $N$, $N \subseteq F$, is a* maximal satisfiable subset (MSS) *of $F$ iff $N$ is satisfiable and for all $f \in F \setminus N$ the set $N \cup \{f\}$ is unsatisfiable.*

**Definition 2** (MCS)**.** *A set $N$, $N \subseteq F$, is a* minimal correction subset (MCS) *of $F$ iff $F \setminus N$ is a MSS of $F$.*

**Definition 3** (MUS)**.** *A set $N$, $N \subseteq F$, is a* minimal unsatisfiable subset (MUS) *of $F$ iff $N$ is unsatisfiable and for all $f \in N$ the set $N \setminus \{f\}$ is satisfiable.*

Note that the minimality/maximality concept used here is a set minimality/maximality, not minimum/maximum cardinality. Therefore, there can be MUSes/MSSes/MCSes with different cardinalities. In general, there can be up to exponentially many MUSes, MSSes, and MCSes, of $F$ w.r.t. $|F|$ (see the Sperner's theorem [36]). We use $\texttt{AMUS}_F$, $\texttt{AMSS}_F$, and $\texttt{AMCS}_F$ to denote the sets of all MUSes, MSSes, and MCSes of $F$, respectively. Moreover, we use $\texttt{UMUS}_F$ and $\texttt{UMCS}_F$ to denote the union of all MUSes and all MCSes of $F$, respectively, and we use $\texttt{IMSS}_F$ to denote the intersection of all MSSes of $F$.

**Example 1.** We demonstrate the concepts on a small example, illustrated in Figure 1a. Assume that we are given a formula $F = \{f_1 = \{x_1\}, f_2 = \{\neg x_1\}, f_3 = \{x_2\}, f_4 = \{\neg x_1, \neg x_2\}\}$. The sets of interests are the following: $\texttt{AMUS}_F = \{\{f_1, f_2\}, \{f_1, f_3, f_4\}\}$, $\texttt{AMSS}_F = \{\{f_2, f_3, f_4\}, \{f_1, f_4\}, \{f_1, f_3\}\}$, $\texttt{AMCS}_F = \{\{f_1\}, \{f_2, f_3\}, \{f_2, f_4\}\}$, $\texttt{UMUS}_F = \texttt{UMCS}_F = F$, and $\texttt{IMSS}_F = \emptyset$.

There is a well-known relationship between $\texttt{AMUS}_F$ and $\texttt{AMCS}_F$ defined in terms of *hitting sets*. Given a collection $\mathcal{S}$ of sets, a hitting set $H$ for $\mathcal{S}$ is a set such that $\forall S \in \mathcal{S}. H \cap S \neq \emptyset$. Especially, a hitting set is *minimal* if none of its proper subsets is a hitting set. Reiter [35] and de Kleer and Williams [19] have shown that $M \in \texttt{AMUS}_F$ iff $M$ is a minimal hitting set of $\texttt{AMCS}_F$. Dually, $N \in \texttt{AMCS}_F$ iff $N$ is a minimal hitting set of $\texttt{AMUS}_F$. Consequently, $\texttt{UMUS}_F = \texttt{UMCS}_F$. Moreover, since $N \in \texttt{AMCS}_F$ iff $F \setminus N \in \texttt{AMSS}_F$, then $\texttt{IMSS}_F = F \setminus \texttt{UMCS}_F = F \setminus \texttt{UMUS}_F$.

**Definition 4** (conflicting clause)**.** *Let $S$ be a satisfiable subset of $F$. A clause $f \in F \setminus S$ is* conflicting *for $S$ iff $S \cup \{f\}$ is unsatisfiable.*

**Definition 5** (critical clause)**.** *Let $U$ be an unsatisfiable subset of $F$. A clause $f \in U$ is* critical *for $U$ iff $U \setminus \{f\}$ is satisfiable.*

Note that if $f$ is conflicting for $S$ then $f$ is conflicting for every satisfiable superset of $S$, and especially for every MSS $S_{mss} \supseteq S$. Furthermore, note that a set $S$ is a MSS iff every $f \in F \setminus S$ is conflicting for $S$. Dually, if $f$ is critical for $U$ then $f$ is critical for every unsatisfiable subset of $U$, and $U$ is a MUS iff every $f \in U$ is critical for $U$.

Another related concept are *backbone literals* of a set $S \subseteq F$, i.e., literals that have to be satisfied by every model of $S$. There is an important connection between backbone literals and conflicting clauses as stated in Observation 1.

**Observation 1.** If $S$ is a satisfiable subset of $F$ and a clause $f \in F \setminus S$ is conflicting for $S$ then every model of $S$ has to falsify $f$. Consequently, the literals $\{\neg l \,|\, l \in f\}$ are backbone literals of every $S'$, $S' \supseteq S$.

Finally, we exploit capabilities of contemporary SAT solvers. Given a set $N \subseteq F$, a contemporary SAT solver is able to provide an *unsat core* of $N$ when $N$ is unsatisfiable, and a *model* $\pi$ of $N$ when $N$ is satisfiable. The unsat core is often small yet not necessarily a minimal unsatisfiable subset of $N$. The model $\pi$, on the other hand, can be used to *extend* $N$ into a larger satisfiable subset of $F$. In particular, the *model extension* $E$ of $N$ w.r.t. $\pi$ and $F$ is the set $\{f \in F \,|\, \pi(f) = 1\}$. Note that $E$ is necessarily satisfiable ($\pi$ is its model), and that $E \supseteq N$. The extraction of models and unsat cores from a SAT solver usually comes with almost no overhead compared to a standard satisfiability query.

## 2.2 Unexplored Subsets

Our MSS enumeration algorithm during its computation gradually determines satisfiability of individual subsets of $F$. *Explored* subsets are those whose satisfiability has been already determined and *unexplored* subsets are the other ones. The algorithm maintains a set `Unexplored` that contains all unexplored subsets. Initially, all proper subsets of $F$ are unexplored, i.e., `Unexplored` $= \mathcal{P}(F) \setminus \{F\}$ (we assume that $F$ is unsatisfiable). Eventually, our algorithm determines satisfiability of every subset of $F$ and, thus, `Unexplored` becomes empty. We further classify unexplored subsets (i.e., elements of `Unexplored`) as follows:

**Definition 6** (s-seed)**.** *A set $N$ is an* s-seed *iff $N$ is unexplored and satisfiable.*

**Definition 7** (u-seed)**.** *A set $N$ is a* u-seed *iff $N$ is unexplored and unsatisfiable.*

Note that if a set $S$ of clauses is satisfiable, then also all subsets of $S$ are satisfiable. Thus, when our algorithm removes a satisfiable $S$ from `Unexplored`, it also removes all subsets of $S$ from `Unexplored`. Dually, when the algorithm removes an unsatisfiable $U$ from `Unexplored`, it also removes all supersets of $U$ from `Unexplored` (since they are necessarily unsatisfiable). Especially, our algorithm maintains the following invariant w.r.t. `Unexplored`:

**Invariant 1.** If a set $S$, $S \subseteq F$, is satisfiable and $S \notin$ `Unexplored`, then for every $N \subseteq S$ it holds that $N \notin$ `Unexplored`. Dually, if a set $U$, $U \subseteq F$, is unsatisfiable and $U \notin$ `Unexplored`, then for every $M \supseteq U$ it holds that $M \notin$ `Unexplored`.

**Observation 2.** If $N$ is an s-seed, then every satisfiable superset of $N$ is also an s-seed. Dually, if $M$ is a u-seed, then every unsatisfiable subset of $M$ is also a u-seed.

*Proof.* Let $N$ be an s-seed and $S$ a satisfiable superset of $N$ that is not an s-seed, i.e., $S \notin$ `Unexplored`. Due to Invariant 1, $N \notin$ `Unexplored` (contradiction). Dually for the u-seed $M$. □

**Observation 3.** If $N$ is an s-seed such that for all $f \in F \setminus N$ the set $N \cup \{f\}$ is explored, then $N$ is a MSS.

*Proof.* $N$ is a MSS if for all $f \in F \setminus N$ the set $N \cup \{f\}$ is unsatisfiable (Definition 1). Assume that there is $f \in F \setminus N$ such that the set $S = N \cup \{f\}$ is satisfiable. Since $S = N \cup \{f\}$ is explored then $N$ has to be also explored (Invariant 1), which contradicts that $N$ is an s-seed. □

**Example 2.** We illustrate the concepts on the set of four clauses from Example 1: $F = \{\{x_1\}, \{\neg x_1\}, \{x_2\}, \{\neg x_1, \neg x_2\}\}$. Fig. 1b shows a possible state of exploration of the power-set. There are two explored unsatisfiable subsets (red with dashed border), seven explored satisfiable subsets (green with solid border), three u-seeds (black with dashed border), and four s-seeds (black with solid border).

Note that given an s-seed $S$, the set `Unexplored` allows us to determine that some clauses from $F \setminus S$ are conflicting for $S$. For instance, in Example 2, we can see that $f_2$ is conflicting for the s-seed $S = \{f_1, f_4\}$ since $S \cup \{f_2\}$ is explored and thus unsatisfiable. We say that $f_2$ is a *minable conflicting* for $S$. Similarly, given a u-seed $U$, we can collect some *minable critical* clauses for $U$. Formally, we define it as follows.

**Definition 8** (minable conflicting)**.** *Let $S$ be an s-seed and $f$ a conflicting clause for $S$. The clause $f$ is a* minable *conflicting clause for $S$ if $S \cup \{f\} \notin$* `Unexplored`*.*

**Definition 9** (minable critical)**.** *Let $U$ be a u-seed and $f$ a critical clause for $U$. The clause $f$ is a* minable *critical clause for $U$ if $U \setminus \{f\} \notin$* `Unexplored`*.*

The exact way we represent (store) and perform operations over the set `Unexplored` is described in Section 4. For now, let us just note that we use a symbolic representation that avoids explicitly storing every single unexplored subset.

# 3 Algorithm

Our MSS enumeration algorithm, called RIME, is based on a schema that we call *seed-grow schema*: to find each single MSS, we find an s-seed $N$ and then we *grow* $N$ into a MSS $N_{mss}$ of $F$ such that $N \subseteq N_{mss} \subseteq F$. The seed-grow scheme has been already used by several MSS enumeration algorithms, e.g., [26, 37, 14]. In general, contemporary seed-grow algorithms identify an s-seed by iteratively picking and checking an unexplored subset for satisfiability, via a SAT solver, until they find an s-seed. Individual seed-grow algorithms vary in *which* and *how many* unexplored subsets are checked for satisfiability. Moreover, the algorithms vary in how exactly they grow the s-seeds. We provide more details on contemporary seed-grow algorithms and other MSS enumeration algorithms in Section 5.

RIME employs a novel growing procedure. Moreover, RIME alternates two novel techniques for finding s-seeds. One technique works on the same principle as the contemporary algorithms do: RIME repeatedly checks unexplored subsets for satisfiability until it finds an s-seed. The novelty is in the choice of unexplored subsets to be checked. Briefly, RIME maintains a *base* $I_F$, $I_F \subseteq F$, and a *search-space* $\mathcal{X}$ defined as $\mathcal{X} =$ `Unexplored` $\cap \{N \mid I_F \subseteq N \subseteq F\}$, i.e., $\mathcal{X}$ consists of those unexplored subsets that contain the whole base. RIME searches for s-seeds in the search-space $\mathcal{X}$. The base is gradually updated in a way ensuring that s-seeds in the search-space can be easily found and grown, and that we eventually find all MSSes.

The other technique for finding s-seeds, called *MSS rotation*, is fundamentally different: we exploit the already explored MSSes to deduce that some unexplored subsets are satisfiable (i.e., s-seeds). The deduction is very cheap and does not involve a usage of a SAT solver. In the following, we gradually provide a thorough description of all parts of RIME.

**Algorithm 1:** RIME

```
1  Unexplored ← P(F) \ {F}
2  I_F ← F
3  while Unexplored ≠ ∅ do
4      (I_F, Unexplored) ← refine(I_F, Unexplored)
5      while Unexplored ∩ {N | I_F ⊆ N ⊆ F} ≠ ∅ do
6          N ← a minimal element of Unexplored ∩ {N | I_F ⊆ N ⊆ F}
7          (sat?, E, K) ← isSAT(N)
8          if sat? then
9              (E_mss, Unexplored) ← grow(E, Unexplored)
10             output E_mss
11             Unexplored ← Unexplored \ {X | X ⊆ E_mss ∨ X ⊇ E_mss}
12             (I_F, Unexplored) ← rotate(E_mss, I_F, Unexplored)
13         else Unexplored ← Unexplored \ {X | X ⊇ K}
```

## 3.1 Main Procedure

The main procedure of RIME is shown in Algorithm 1. Initially, the base $I_F$ is set to $F$ and `Unexplored` is set to $\mathcal{P}(F) \setminus \{F\}$ (we assume that $F$ is unsatisfiable). The rest of the algorithm is formed by two nested while-loops. At the start of each iteration of the outer loop, the algorithm updates the base $I_F$ via a procedure `refine`. Moreover, `refine` can possibly identify some MSSes. Subsequently, in the nested loop, the algorithm finds all unexplored MSSes in the search-space $\mathcal{X} = \texttt{Unexplored} \cap \{N \mid I_F \subseteq N \subseteq F\}$. In particular, each iteration of the nested loop starts by picking a minimal element $N$ of $\mathcal{X}$ where *minimal* means that $N \in \mathcal{X}$ and for all $f \in N$ it holds that $N \setminus \{f\} \notin \mathcal{X}$. Subsequently, $N$ is processed by a procedure `isSAT` that determines the satisfiability of $N$ (via a SAT solver) and moreover identifies either an unsat core $K$ or a model extension $E$ of $N$. If $N$ is unsatisfiable, the algorithm removes all supersets of the unsat core $K$ from `Unexplored` (since they are all unsatisfiable). In the other case, when $N$ is satisfiable, the algorithm grows the model extension $E$ into a MSS $E_{mss}$ using a procedure `grow`. Moreover, the algorithm removes all subsets and all supersets of $E_{mss}$ from `Unexplored`, since none of them can be another MSS. Then, RIME applies on $E_{mss}$ the *MSS Rotation technique* (denoted by `rotate`) that tends to identify additional MSSes. The MSS rotation can also reduce the set `Unexplored` and update the base $I_F$. The inner loop terminates once $\mathcal{X} = \emptyset$. The outer loop terminates once $\texttt{Unexplored} = \emptyset$.

Detailed description of how the procedures `refine`, `rotate`, and `grow` work are provided in Sections 3.2, 3.3 and 3.4, respectively. For now, let us state that in all the procedures we remove elements from `Unexplored` only in two situations. First, every time RIME identifies a MSS, it removes the MSS together with all subsets and all supersets of the MSS from `Unexplored`. Second, RIME can remove an unsatisfiable $U$, $U \subseteq F$, together with all supersets of $U$ from `Unexplored`. Thus, no MSS can be removed from `Unexplored` without being explicitly identified. Furthermore, Observation 4 holds.

**Observation 4.** The outer loop of Algorithm 1 terminates iff all MSSes and all MUSes have been explored.

*Proof.* $\Rightarrow$: The outer loop terminates when all subsets of $F$ are explored, and thus, especially, all MSSes and MUSes are explored.

$\Leftarrow$: Every satisfiable subset of $F$ is a subset of a MSS of $F$ and symmetrically, every unsatisfiable subset of $F$ is a superset of a MUS of $F$. In a conjuncture with Invariant 1, the set `Unexplored` becomes empty once all MSSes and all MUSes become explored. □

## 3.2 The Base and the Search-Space

Here we describe how we form and maintain the base $I_F$ and thus the search-space $\mathcal{X} = \texttt{Unexplored} \cap \{N \mid I_F \subseteq N \subseteq F\}$. To build a suitable base, we make use of the intersection $\texttt{IMSS}_F$ of all MSSes of $F$. Imagine that we can cheaply compute $\texttt{IMSS}_F$ and that we set $I_F$ to $\texttt{IMSS}_F$. Since every MSS of $F$ is a superset of $\texttt{IMSS}_F$, the induced search-space $\mathcal{X} = \texttt{Unexplored} \cap \{N \mid \texttt{IMSS}_F \subseteq N \subseteq F\}$ would contain all unexplored MSSes. Furthermore, if $|\texttt{IMSS}_F|$ is relatively large w.r.t. the (average) size of MSSes, then s-seeds in such $\mathcal{X}$ are relatively close to MSSes and thus should be easy to grow. Unfortunately, computing $\texttt{IMSS}_F$ is often practically intractable in a reasonable time [28]. Thus, RIME maintains an over-approximation of $\texttt{IMSS}_F$ as the base $I_F$, i.e., $\texttt{IMSS}_F \subseteq I_F \subseteq F$. Initially, we set $I_F$ to $F$ (Algorithm 1, line 2), and in each call of the procedure `refine` (Algorithm 1, line 4), we reduce $I_F$ by removing at least a single clause of $I_F \setminus \texttt{IMSS}_F$ from $I_F$. Eventually, $I_F$ becomes $\texttt{IMSS}_F$, and thus eventually the search-space $\mathcal{X}$ will contain all so far unexplored MSSes.

The procedure $\texttt{refine}(I_F^i, \texttt{Unexplored}^i)$ is based on the following three observations; we use the superscript "$^i$" to distinguish the input values of $I_F$ and `Unexplored`.

**Lemma 1.** Whenever Algorithm 1 invokes the procedure `refine`, it holds that $\mathcal{X} = \{N \mid I_F^i \subseteq N \subseteq F\} \cap \texttt{Unexplored}^i = \emptyset$.

*Proof.* At the first iteration of the outer loop, $\texttt{Unexplored}^i = \mathcal{P}(F) \setminus \{F\}$ and $I_F^i = F$, thus $\mathcal{X} = \emptyset$. In other iterations, the claim follows from the condition of the inner loop. □

**Lemma 2.** Let $M$ be a MSS of $F$ such that $M \in \texttt{Unexplored}^i$. Then $\texttt{IMSS}_F \subseteq I_F^i \cap M \subsetneq I_F^i$.

*Proof.* From the pre-condition $\mathcal{X} = \emptyset$ (Lemma 1), we have $I_F^i \setminus M \neq \emptyset$, thus $I_F^i \cap M \subsetneq I_F^i$. Furthermore, since both $I_F^i$ and $M$ are supersets of $\texttt{IMSS}_F$, we have $\texttt{IMSS}_F \subseteq I_F^i \cap M$. □

**Lemma 3.** Let $M$ be a MUS of $F$ such that $M \in \texttt{Unexplored}^i$. Then $\texttt{IMSS}_F \subseteq I_F^i \setminus M \subsetneq I_F^i$.

*Proof.* From the minimal hitting set duality between MUSes and MCSes, we know that for each $f \in M$, there has to exist a MCS $L$ such that $f \in L$ and its complementing MSS $\overline{L}$ such that $f \notin \overline{L}$. Thus, we have that $\texttt{IMSS}_F \subseteq I_F^i \setminus M$. Furthermore, since $M \in \texttt{Unexplored}^i$ and from the pre-condition $\mathcal{X} = \emptyset$ (Lemma 1), we have $I_F^i \setminus M \neq \emptyset$, thus $I_F^i \setminus M \subsetneq I_F^i$. □

In other words, every unexplored MSS or MUS allow us to reduce $I_F$. Moreover, from Observation 4, we know that there is at least one unexplored MSS or MUS when `refine` is invoked. The procedure `refine` is shown in Algorithm 2. Each iteration of the algorithm starts by picking a *maximal* unexplored subset, i.e., a set $T \in \texttt{Unexplored}$ such that for each $f \in F \setminus T$ it holds that $T \cup \{f\} \notin \texttt{Unexplored}$. Then, $T$ is checked for satisfiability via the procedure `isSAT`. If $T$ is unsatisfiable, `isSAT` returns an unsat core $K$ of $T$. Subsequently, a procedure `shrink` is used to find a MUS $K_{mus}$ of $K$, `Unexplored` is reduced by removing all supersets of $K_{mus}$ from it, and $I_F$ is reduced to $I_F \setminus K_{mus}$. In the other case, when $T$ is satisfiable, it is guaranteed that $T$ is an MSS of $F$ (Observation 3). The algorithm reduces $I_F$ to $I_F \cap T$, and removes all subsets and all supersets of $T$ from `Unexplored`. Subsequently, $T$ is processed by the procedure `rotate` that can identify additional MSSes and further reduce $I_F$ and `Unexplored`. The algorithm terminates either when a first MUS is found or when $\texttt{Unexplored} = \emptyset$.

**Algorithm 2:** `refine`$(I_F, \texttt{Unexplored})$

```
1  while Unexplored ≠ ∅ do
2      T ← a maximal element of Unexplored
3      (sat?, K) ← isSAT(T)
4      if sat? then
5          output T                                          // T is a MSS
6          Unexplored ← Unexplored \ {X | X ⊆ T ∨ X ⊇ T}
7          I_F ← I_F ∩ T
8          (I_F, Unexplored) ← rotate(T, I_F, Unexplored)
9      else
10         K_mus ← shrink(K, getMinableCriticals(K, Unexplored))
11         Unexplored ← Unexplored \ {X | X ⊇ K_mus}
12         I_F ← I_F \ K_mus
13         break
14 return (I_F, Unexplored)
```

The procedure `shrink` that finds a MUS of $K$ can be implemented via any single MUS extraction algorithm, e.g., [3, 11, 8, 30]. Since contemporary single MUS extractors can largely benefit from a prior knowledge of critical clauses, we also provide the extractor with the set of clauses that are minable critical for $K$ (denoted by `getMinableCriticals` in Algorithm 2).

Finally, we provide some additional insight. First, one might ask why we allow Algorithm 2 to identify multiple MSSes even though finding just a single MSS would be enough to reduce $I_F$. The reason is that each of these MSSes is identified using just a single satisfiability check, i.e., very cheaply. Second, note that since RIME grows s-seeds that are supersets of $I_F$, the fact that we gradually reduce the size of $I_F$ implies that we gradually reduce the expected size of the s-seeds. It is often the case that the larger an s-seed is, the easier it is to grow the s-seed (see Section 3.4). Thus, gradually reducing $I_F$ to $\texttt{IMSS}_F$ give us more suitable (larger) s-seeds than if we set $I_F$ to $\texttt{IMSS}_F$ immediately at the beginning of the computation.

## 3.3 MSS Rotation

Here we describe our *MSS Rotation* technique that, based on a MSS $N$ of $F$, attempts to identify additional unexplored MSSes. Moreover, it attempts to reduce the sets `Unexplored` and $I_F$. From the high-level view, MSS rotation follows the seed-grow scheme, i.e., it identifies s-seeds and grows them to MSSes. The uniqueness of MSS rotation lies in the way it finds the s-seeds. Instead of first picking an unexplored subset and then checking the unexplored subset for satisfiability, MSS rotation first finds a satisfiable subset and then checks the subset for being unexplored. Such an approach has two significant advantages. First, it is much easier to check a subset for being unexplored than to find an unexplored subset (see Section 4). Second, MSS rotation finds the satisfiable subset via a cheap deduction technique instead of employing a SAT solver. The key deduction idea is summarized in Lemma 4.

**Lemma 4.** Let $M$ be a MSS of $F$, $f$ a clause such that $f \in F \setminus M$, and $l$ a literal of $f$. Then, the set $S = (M \cup \{f\}) \setminus \{g \in M \mid \neg l \in g\}$ is satisfiable.

*Proof.* Let $\pi$ be a model of $M$ and $\pi_l$ a truth assignment that originates from $\pi$ by flipping the assignment to the variable of $l$. Since $\pi$ satisfies the whole $M$, then $\pi_l$ satisfies at least those

**Algorithm 3:** `rotate(`$N$`,`$I_F$`,Unexplored)`

```
1  rotationQueue ← ⟨N⟩
2  while rotationQueue is not empty do
3      M ← rotationQueue.dequeue()
4      for f ∈ F \ M do
5          for l ∈ f do
6              S ← (M ∪ {f}) \ {g ∈ M | ¬l ∈ g}
7              if |M \ S| > threshold then continue
8              if S ∈ Unexplored then
9                  (S_mss, Unexplored) ← grow(S, Unexplored)
10                 output S_mss
11                 Unexplored ← Unexplored \ {X | X ⊆ S_mss ∨ X ⊇ S_mss}
12                 I_F ← I_F ∩ S_mss
13                 rotationQueue.enqueue(S_mss)
14 return (I, Unexplored)
```

clauses of $M$ that does not contain $\neg l$. Moreover, since $l \in f$ then $\pi_l$ satisfies also $f$. □

The MSS rotation technique is shown in Algorithm 3. The algorithm maintains a queue *rotationQueue* of MSSes. Initially, the queue contains the input MSS $N$. In each iteration, the algorithm dequeues a MSS $M$ from the queue and uses it to find new MSSes. In particular, for each $f \in F \setminus M$ and each $l \in f$, the algorithm constructs a satisfiable set $S$ based on Lemma 4 and checks if $S$ is unexplored. If $S$ is unexplored, then $S$ is grown to a MSS $S_{mss}$ and the MSS is used to reduce the sets `Unexplored` and $I_F$. Moreover, $S_{mss}$ is added to the queue and thus eventually used to possibly identify additional MSSes.

We employ one additional heuristic in the algorithm: instead of checking every $S$ for being an s-seed, we skip every $S$ such that $|M \setminus S| > \mathit{threshold}$ where $\mathit{threshold} \geq 1$ (line 7 in the algorithm). There are two motivations for this heuristic. First, the smaller $S$ is, the more-likely $S$ is a subset of some already explored MSS, i.e., the more-likely is $S$ explored. Second, it is generally easier to grow larger s-seeds than smaller s-seeds, thus we tend to find larger s-seeds. The most suitable value of *threshold* varies for different kinds of benchmarks; thus the value of *threshold* can be specified by the user of our algorithm.

**Model rotation** There is a technique called *model rotation* [10] that is similar to MSS rotation; in fact, we named the MSS rotation after the model rotation. Model rotation is used in contemporary single MUS extraction tools (e.g., [11, 8, 3]) and it serves for identifying critical clauses of a given unsatisfiable formula. In particular, let $N$ be an unsatisfiable formula, $f$ a critical clause for $N$ and $\pi$ a *model* of $N \setminus \{f\}$. Model rotation *rotates* the model $\pi$ into a truth assignment $\pi_x$ by flipping an assignment to a single variable $x \in \mathit{Vars}(\{f\})$. Clearly, $\pi_x$ satisfies $\{f\}$ (due to the variable $x$) and $\pi_x$ does not satisfy at least one clause $g \in N$ since $N$ is unsatisfiable. If it is the case that there is exactly one such $g$ then $g$ is a critical clause for $N$. Such a *rotatation* $\pi_x$ of $\pi$ is constructed for every $x \in \mathit{Vars}(\{f\})$ and each such $\pi_x$ can potentially point out a different critical clause. Moreover, the model rotation can be recursively applied on each newly identified critical clause.

In our Lemma 4, the set $M \cup \{f\}$ is unsatisfiable and $\{f\}$ is critical for $M$, thus, we have a similar starting point as in the case of model rotation ($N = M \cup \{f\}$). However, MSS rotation

**Algorithm 4:** `simpleMSSExtraction(`$N$`)`

```
1 C ← ∅
2 while F \ (N ∪ C) ≠ ∅ do
3 |   f ← pick a clause from F \ (N ∪ C)
4 |   (sat?, E) ← isSAT(N ∪ {f}, ⋃_{f∈C} {¬l | l ∈ f})
5 |   if sat? then N ← E
6 |   else C ← C ∪ {f}
7 return N
```

does not take a model as an input. Moreover, model rotation attempts to identify a satisfiable subset $S$ of the input set $N$ such that $S = N \setminus \{g\}$ for some $g \in N$, i.e., $|N \setminus S| = 1$. On the other hand, in MSS rotation, we seek for a satisfiable subset $S$ of $N = M \cup \{f\}$ such that $|M \setminus S| \leq \mathit{threshold} \geq 1$.

## 3.4 Grow

In this section, we describe the procedure `grow` that takes as an input an s-seed $N$ and returns an unexplored MSS $N_{mss}$ of $F$ such that $N \subseteq N_{mss}$. The procedure is based on a simple, well-known, single MSS extraction algorithm. We first describe the base solution and then introduce several novel improvements to the base solution.

**Base Solution** Recall that a set $N$ is a MSS of $F$ if every $f \in F \setminus N$ is conflicting for $N$. Also, recall that if a clause $f$ is conflicting for $N$, than the literals $\{\neg l \mid l \in f\}$ are backbone literals for $N$. The base solution is shown in Algorithm 4. It maintains two sets: the input set $N$ and a set $C$ of clauses that are known to be conflicting for $N$. Initially, $C = \emptyset$. In each iteration, the algorithm picks a clause $f \in F \setminus (N \cup C)$ and checks if $f$ is conflicting for $N$ by checking $N \cup \{f\}$ for satisfiability via the procedure `isSAT`. To speed up the satisfiability query, `isSAT` takes as an input also the set $\bigcup_{f \in C} \{\neg l \mid l \in f\}$ of backbone literals that can be obtained from $C$ (prior knowledge of backbone literals can be very beneficial for a SAT solver). If $N \cup \{f\}$ is satisfiable, then `isSAT` returns a model extension $E$ of $N \cup \{f\}$ and $N$ is enlarged to $E$. In the other case, when $N \cup \{f\}$ is unsatisfiable, the clause $f$ is added to $C$. The computation terminates once $F \setminus (N \cup C) = \emptyset$. The invariant of the algorithm is that $N$ is satisfiable and all clauses in $C$ are conflicting for $N$. The algorithm performs up to $|F \setminus N|$ satisfiability checks (due to the use of model extensions, the number can be lower).

Algorithm 4 forms a basis of several contemporary single MSS extraction algorithms (see [27]). These algorithms extensively exploit domain specific properties of the Boolean CNF domain and are very efficient. Thus, instead of developing a custom MSS extractor, we might use as a black-box subroutine one of the existing solutions. However, the single extractors are tailored for finding only a single MSS and consequently do not (and cannot) fully exploit information that we can obtain from the overall MSS enumeration algorithm. Thus, we propose our custom single MSS extraction algorithm that works in a synergy with the overall MSS enumeration.

**Key Observations** Our approach is based mainly on two observations. First, we can use `Unexplored` to collect minable conflicting clauses for an s-seed $N$. The second observation concerns a concept of *conflict extension* that we define as follows.

**Definition 10** (conflict extension)**.** *Let $N$ be an s-seed, $C$ a set of conflicting clauses for $N$, and $\mathit{backs} = \bigcup_{f \in C} \{\neg l \mid l \in f\}$ the set of all backbone literals for $N$ that can be obtained from $C$.*

**Algorithm 5:** `enlarge`$(N, C,$ `Unexplored`$)$

```
1 while True do
2     E ← conflictExtension(N, C)
3     if E = N then return (N, C)
4     N ← E
5     C′ ← getMinable(N, Unexplored)
6     if C = C′ then return (N, C)
7     C ← C′
```

*The* conflict extension *of $N$ w.r.t. $F$ and $C$ is the set $E = N \cup \{g \in F \setminus N \mid backs \cap Lits(g) \neq \emptyset\}$ where $Lits(g)$ are literals of the clause $g$.*

**Lemma 5.** Let $N$ be an s-seed, $C$ a set of conflicting clauses for $N$, and $E$ the conflict extension of $N$ w.r.t. $F$ and $C$. Then $E$ is an s-seed such that $N \subseteq E$.

*Proof.* Every model of $N$ has to satisfy all the backbone literals *backs* and consequently also every clause $g \in F$ that contains at least one of the backbone literals, thus $E$ is satisfiable. Since $N$ is an s-seed, $E \supseteq N$ is an s-seed (Observation 2). □

The ability to mine conflicting clauses and the concept of conflict extension can be very powerful if we combine them together. In particular, assume that we are given an s-seed $N$ and the set $C$ of clauses that are minable conflicting for $N$. We can use the conflict extension to enlarge $N$ based on $C$. In turns, there might arise additional minable conflicting clauses for $N$, and so on until a fix-point is reached. We properly describe this functionality in the procedure `enlarge` shown in Algorithm 5. The input of the algorithm is an s-seed $N^i$, a set $C^i$ of conflicting clauses for $N^i$, and the set `Unexplored`. The output of the procedure are sets $N^o$, $C^o$ such that $N^o$ is an s-seed, $C^o$ is a set of conflicting clauses for $N^o$, $N^o \supseteq N^i$, and $C^o \supseteq C^i$. The algorithm works iteratively. In each iteration the algorithm computes the conflict extension $E$ of $N$ based on $C$, and then enlarges $N$ to $E$. Subsequently, the set $C'$ of minable conflicting clauses for (the enlarged) $N$ is computed, and $C$ is enlarged to $C'$. The algorithm terminates either once the model extension does not enlarge $N$ anymore or once we are not able to collect any new minable conflicting clauses.

**Algorithm** Our growing procedure is shown in Algorithm 6. It takes as an input an s-seed $N$ together with the set `Unexplored`, and it returns a reduced set `Unexplored` together with an unexplored MSS $N_{mss}$ of $F$ such that $N \subseteq N_{mss}$.

The computation starts by collecting the set $C$ of clauses that are minable conflicting for $N$. Subsequently, the procedure `enlarge` (Algorithm 5) is used to enlarge both $C$ and $N$. The main part of the algorithm is formed by a while loop. In each iteration, the algorithm picks a clause $f \in F \setminus (N \cup C)$ and checks $N \cup \{f\}$ for satisfiability via the procedure `isSAT`. To make the check more efficient, we pass to `isSAT` the set $\bigcup_{f \in C}\{\neg l \mid l \in f\}$ of backbone literals that can be obtained from $C$. If $N \cup \{f\}$ is satisfiable, then we obtain its model extension $E$ and enlarge $N$ to $E$. Consequently, there might emerge additional minable conflicting clauses for $N$, thus we re-collect them and add them to $C$. In the other case, when $N \cup \{f\}$ is unsatisfiable, we obtain an unsat core $K$ of $N \cup \{f\}$ from the SAT solver and we remove all supersets of $K$ from `Unexplored`. Also, we add $f$ to $C$ since $f$ is conflicting for $N$. At the end of the iteration, we use the procedure `enlarge` to possibly further enlarge $N$ and $C$.

**Algorithm 6:** `grow`$(N, \texttt{Unexplored})$

```
1  C ← getMinableConflicts(N, Unexplored)
2  N, C ← enlarge(N, C, Unexplored)
3  while F \ (N ∪ C) ≠ ∅ do
4      f ← pick a clause from F \ (N ∪ C)
5      (sat?, E, K) ← isSAT(N ∪ {f}, ⋃_{f∈C} {¬l | l ∈ f})
6      if sat? then
7          N ← E
8          C ← C ∪ getMinableConflicts(N, Unexplored)
9      else
10         C ← C ∪ {f}
11         Unexplored ← Unexplored \ {X | X ⊇ K}
12     N, C ← enlarge(N, C, Unexplored)
13 return (N, Unexplored)
```

Same as in the case of the base solution (Algorithm 4), the invariant of the algorithm is that $N$ is an s-seed, $C$ is a set of conflicting clauses for $N$, and $C \cap N = \emptyset$. The algorithm terminates once $C \cup N = F$, thus the final $N$ is an unexplored MSS of $F$. In the worst case, the algorithm performs $|F \setminus N|$ satisfiability checks, i.e., there is no asymptotic improvement over the base solution. Yet, we have experimentally observed that the algorithm usually performs much fewer satisfiability checks and in some cases even no checks at all (see Section 6). This is mainly due to two reasons. First, RIME grows s-seeds that are usually very close to the resultant MSSes. Second, the procedure `enlarge` can often significantly enlarge both $N$ and $C$.

# 4 Operations Over Unexplored Subsets

To maintain the set `Unexplored`, we adopt a symbolic representation that was first used in the algorithm MARCO [25, 32] and subsequently adopted by many other MSS or MUS enumeration algorithms (e.g., [3, 14, 31, 17]). Given a formula $F = \{f_1, \ldots, f_n\}$, we introduce a set $A = \{a_1, \ldots, a_n\}$ of Boolean variables. Note that each truth assignment to $A$ one-to-one-maps to a subset of $F$. To represent `Unexplored`, we maintain a pair of Boolean formulas, $map^+$ and $map^-$, over $A$ such that each model of the conjunction $map^+ \wedge map^-$ one-to-one-maps to an element of `Unexplored`. The formulas are maintained as follows:

- To represent $\texttt{Unexplored} = \mathcal{P}(F)$, we set the formulas to $map^+ = map^- = \top$.
- To remove a set $N \subseteq F$ and all of its subsets from `Unexplored`, we add to $map^+$ the clause $\bigvee_{f_i \notin N} a_i$.
- Dually, to remove a set $N \subseteq F$ and all of its supersets from `Unexplored`, we add to $map^-$ the clause $\bigvee_{f_i \in N} \neg a_i$.

To get an arbitrary element of `Unexplored`, we can ask a SAT solver for a model of $map^+ \wedge map^-$. However, in RIME, we work with two specific types of unexplored subsets. First, in Algorithm 2, we require a *maximal element* of `Unexplored`, i.e., a set $T \in \texttt{Unexplored}$ such that for every $f \in F \setminus T$ it holds that $T \cup \{f\} \notin \texttt{Unexplored}$. To get such $T$, we ask a SAT solver for a *maximal model* of $map^+ \wedge map^-$. Second, in Algorithm 1, we require a *minimal element* $N$ of $\mathcal{X} = \texttt{Unexplored} \cap \{N \mid I_F \subseteq N \subseteq F\}$, i.e., a set $N \in \mathcal{X}$ such that for every

$f \in N$ it holds that $N \setminus \{f\} \notin \mathcal{X}$. To get such $N$, we instruct the SAT solver to fix the values of the variables $\{x_i \mid f_i \in I_F\}$ to $\top$ and ask it for a *minimal model* of $map^+ \wedge map^-$.

One of the contemporary SAT solvers that allows the user to fix values of some variables and can provide minimal and maximal models is miniSAT [20] (the minimal/maximal models are enforced by setting the default values (polarity) of variables to false/true). In our implementation, we employ miniSAT in an incremental manner, i.e., we incrementally build the formula $map^+ \wedge map^-$ and we let miniSAT to internally simplify the formula whenever possible.

As for collecting minable conflicting clauses, recall that given an s-seed $T$ and a clause $f \in F \setminus T$, the clause $f$ is minable conflicting for $T$ iff $T \cup \{f\} \notin$ `Unexplored`. Speaking in terms of our symbolic representation, $f$ is minable conflicting for $T$ iff $T \cup \{f\}$ does not satisfy $map^+ \wedge map^-$. To collect the set $C$ of all minable conflicting clauses for $T$, we check for every $f \in F \setminus T$ whether $T \cup \{f\}$ satisfies $map^+ \wedge map^-$. To perform such checks efficiently, we exploit one more observation. In particular, observe that the information contained in $map^+$ is irrelevant for mining conflicting clauses. Intuitively, $map^+$ contains only positive literals and thus *requires a presence* of clauses. Since $T$ satisfies $map^+$ ($T \in$ `Unexplored` ), then $T \cup \{f\}$ necessarily also satisfies $map^+$. Consequently, $T \cup \{f\} \in$ `Unexplored` iff $T \cup \{f\}$ satisfies $map^-$.

Finally, collecting minable critical clauses for a u-seed is a dual problem to collecting minable conflicting clauses for an s-seed, thus we handle it in the dual manner.

# 5 Related Work

Whenever a MSS enumeration algorithm searches for a MSS $N$, it needs to deal with two tasks: (1) it has to guarantee that $N$ is indeed a MSS, and (2) it has to guarantee that $N$ is so far unexplored MSS. Based on these tasks, we can divide existing algorithms into two categories.

Algorithms from one category deal with the two tasks separately and are based on the seed-grow scheme. The algorithms start by finding an s-seed, i.e., with task (2), and then they grow the s-seed into a MSS, i.e., perform the task (1). Perhaps the most famous contemporary seed-grow algorithm is MARCO [26]. To find an s-seed, MARCO is repeatedly picking and checking for satisfiability (via a SAT solver) a minimal unexplored subset until an s-seed is found. Since satisfiable subsets of $F$ are naturally more concentrated among the smaller subsets, MARCO often needs only few satisfiability checks to find the s-seed. However, s-seeds found among minimal unexplored subsets are usually very small and thus potentially hard to grow. As for the growing itself, the authors of MARCO in their paper state that any existing (or even future) single MSS extractor can be used to implement the growing. In fact, one can incorporate the growing procedure that we presented in this paper into MARCO. The most important difference between MARCO and RIME is thus in the way the algorithms find the s-seeds. There are also some earlier algorithms, e.g., DAA [5] and PDDS [37], that search for s-seeds among minimal unexplored subsets, however, the algorithms use an inefficient representation of the unexplored subsets. Whereas MARCO uses the symbolic formula $map^+ \wedge map^-$, DAA and PDDS use a less efficient representation based on minimal hitting sets.

Algorithms from the other group deal with the tasks (1) and (2) simultaneously. Similarly as MARCO and RIME, the algorithms use a variation of the formula $map^+ \wedge map^-$ to carry information about unexplored subsets. However, instead of using the formula in a separate manner, they combine $map^+ \wedge map^-$ with the formula $F$ into a single Boolean formula $G$. Consequently, the task of checking whether a subset $N$ of $F$ is both satisfiable and unexplored, i.e., an s-seed, can be done via a single SAT solver query. To find each single MSS, the algorithms perform several such queries, and every time a new MSS is found, the formula $G$ is modified to exclude the MSS from further computation. To the best of our knowledge, the current

state-of-the-art MSS enumerators from this category are MCSLS [27] and FLINT [31].

There have been proposed several single MSS extraction algorithms, e.g., [21, 2, 23, 29], that can be often used as a subroutine of MSS enumeration algorithms, and several techniques [33, 34] for caching results of SAT queries that naturally emerge during single or multiple MSS extraction. Finally, there are many tools for MUS enumeration (e.g., [3, 4, 14, 31, 17, 18, 16]). Due to the minimal hitting set duality between MUSes and MCSes [35, 19], many of existing MUS enumeration algorithms can be in some extent dualized for a MSS enumeration.

# 6 Experimental Evaluation

We implemented RIME in C++ with a help of miniSAT [20] as a SAT solver and a single MUS extractor muser2 [11] to perform the shrinking in Algorithm 2. Our tool is publicly available at https://github.com/jar-ben/rime. Here we provide results of our experimental evaluation. Besides evaluating RIME, we also provide a comparison with three contemporary MSS enumeration tools: MARCO[1] [26], MCSLS[2] [27], and FLINT[3] [31].

We focus on three criteria in the comparison: (1) the number of identified MSSes within a given time limit, (2) the median time to identify individual MSSes, and (3) the median number of performed satisfiability checks to identify individual MSSes.

We used a collection of 383 unsatisfiable CNF benchmarks that were taken from the SAT Competition[4]. The benchmarks range in their size from 70 to 16 million clauses and use from 26 to 4.4 million variables. All experiments were run using a time limit of 3600 seconds and computed on an AMD EPYC 7371 16-Core Processor, 1 TB memory machine running Debian Linux 4.19.67-2. The value of *threshold* in Algorithm 3 was set to 10. Complete results of the evaluation are available at: https://www.fi.muni.cz/~xbendik/research/rime.

## 6.1 Number of Identified MSSes

Due to possibly exponentially many MSSes in a benchmark, the complete MSS enumeration is generally intractable. Moreover, even producing a single MSS can be intractable for hard instances. Only in case of 31 benchmarks all the algorithms finished the computation. In case of 6 benchmarks no algorithm found even a single MSS within the time limit.

Figure 2 provides 3 scatter plots that pairwise compare RIME with its competitors on individual benchmarks. Each point in the plot represents the result achieved by the two compared algorithms on one particular benchmark; one algorithm determines the position on the vertical axis and the other one the position on the horizontal axis. There are two types of points: (1) red points show the number of identified MSSes within first 600 seconds and (2) blue points show the number of identified MSSes within 3600 seconds. We provide results achieved within 600 and 3600 seconds to demonstrate how stable is the efficiency of the algorithms with increasing time limit. Note that the plots are in a log-scale and hence cannot show points with a zero coordinate. Therefore, we lifted the points with a 0 coordinate to the 1st coordinate, i.e., the points that are exactly on the x-axis or on the y-axis show benchmarks where one of the algorithms found either only a single MSS or no MSS at all. Moreover, note that each scatter plot contains three additional pairs of numbers that are above/right of/in the top-right corner of the plot; these numbers represent the number of points that are above/below/on the diagonal,

[1] https://sun.iwu.edu/~mliffito/marco/
[2] The tool was kindly provided to us by its authors
[3] The tool was kindly provided to us by its authors
[4] http://www.satcompetition.org/

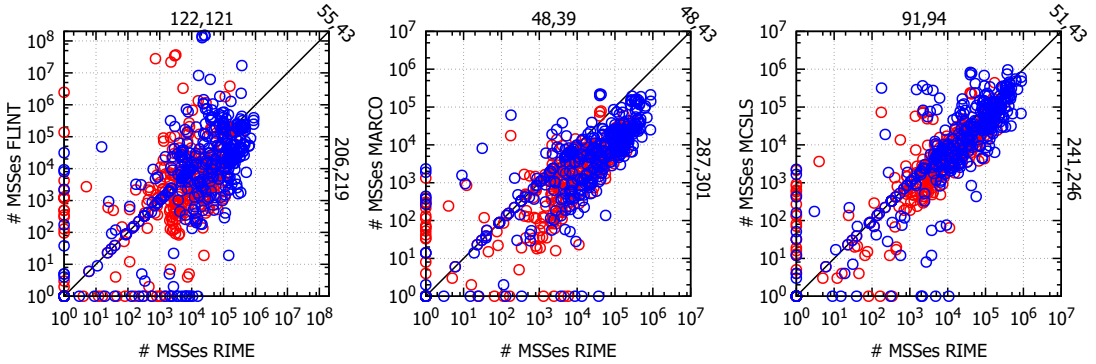



Figure 2: Scatter plots comparing the number of identified MSSes within 600 seconds (red points) and within 3600 seconds (blue points).

| | ranked 1st | ranked 2nd | ranked 3rd | ranked 4th | average ranking |
|---|---|---|---|---|---|
| RIME | 208 \| 227 | 117 \| 86 | 30 \| 42 | 28 \| 28 | 1.68 \| 1.66 |
| FLINT | 145 \| 138 | 50 \| 57 | 87 \| 97 | 101 \| 91 | 2.38 \| 2.37 |
| MCSLS | 107 \| 103 | 164 \| 174 | 100 \| 86 | 12 \| 20 | 2.04 \| 2.06 |
| MARCO | 48 \| 46 | 35 \| 26 | 120 \| 122 | 180 \| 189 | 3.13 \| 3.19 |

Table 1: Overall rankings of evaluated algorithms after 600 and 3600 seconds.

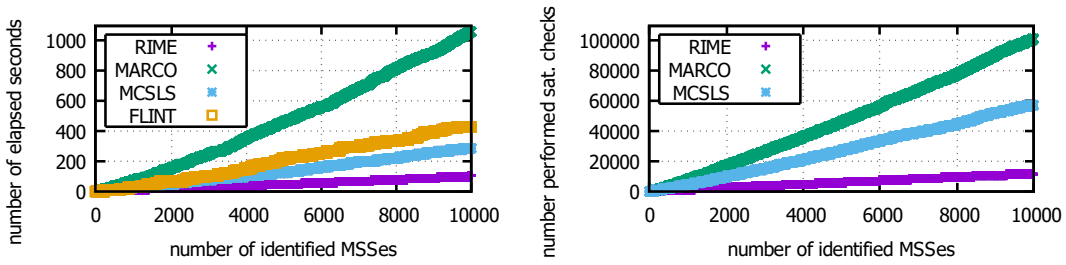
respectively. The first number of a pair corresponds to the red points, and the second number to the blue points. For example, after 3600 seconds, RIME found more/less/equal number of MSS than FLINT in case of 219/121/43 benchmarks, respectively.

Besides the pair-wise comparison of the algorithms, we also provide an overall *ranking* of the algorithms on individual benchmarks. In particular, assume that for a benchmark B both RIME and FLINT found 100 MSSes, MCSLS found 80 MSSes, and MARCO found 50 MSSes. In such a case, RIME and FLINT share the 1st (best) rank for B , FLINT is 3rd, and MARCO is on the 4th position. Table 1 shows a summary of this ranking. In particular, for each algorithm, the table shows the average ranking of the algorithm w.r.t. all benchmarks, and also the number of benchmarks where the algorithm ranks as 1st, 2nd, 3rd, and 4th, respective. In each cell of the table, we show a pair $A|B$ of numbers that correspond to the ranking after 600 seconds ($A$) and after 3600 seconds ($B$).

## 6.2 Time and Checks per MSS

Here, we examine the number of elapsed seconds and the number of performed satisfiability checks to identify each subsequent MSS. These numbers naturally differ for individual benchmarks, thus we focus on median values. The plot in Figure 3a shows the median number of elapsed seconds required to identify the first 10000 MSSes. A point with coordinates $(x, y)$ states that the median of elapsed seconds to output the first $x$ MSSes is $y$. We used only 116 benchmarks to compute the medians since only in those benchmarks all the algorithms found at least 10000 MSSes. The figure shows that all the algorithms produce individual MSSes in a relatively steady rate. The median amount of elapsed time to find the 10000th MSS by RIME, MCSLS, FLINT, and MARCO was 101, 290, 434, and 1062 seconds, respectively.

MSS enumeration naturally subsumes checking subsets of the input Boolean formula for




(a) time per MSS (b) sat. checks per MSS

Figure 3: The median number of elapsed seconds (Fig. 3a) and the median number of performed sat. checks (Fig. 3b) to output the first 10000 MSSes.

satisfiability. Based on our experience, performing these checks is the most expensive part of a MSS enumeration algorithm. In Figure 3b, we show the median number of satisfiability checks performed to identify the first 10000 MSSes (computed from the 116 benchmarks). Unfortunately, we were unable to measure the number of checks for FLINT since its authors provided us only with a binary version of the tool that does not provide this information. We can see that the median number of performed checks to output the 10000th MSS by RIME, MCSLS, and MARCO was 11836, 57813, and 101464, respectively. Thus, to find each single MSS, RIME, MCSLS, and MARCO performed on average around 1.18, 5.8, and 10.15 satisfiability checks.

Remarkably, in case of 60 benchmarks, RIME performed fewer satisfiability checks than what was the number of identified MSSes. This was achieved mainly due to our novel techniques of conflict extension and MSS rotation. On average w.r.t. all the 383 benchmarks, RIME identified 88 percent of all s-seeds via the MSS rotation, i.e., without a single satifiability check. Moreover, in case of 11 percent of all grows, RIME did not perform even a single satifiability check. We conclude that it is the frugality of RIME w.r.t. the number of performed satisfiability checks what allowed it to so substantially outperform all of its competitors. RIME found significantly more MSSes than its competitors on a majority of benchmarks; the difference was often several orders of magnitude. Moreover, in terms of median values, RIME is three times faster than its closest competitor MCSLS, four times faster than FLINT, and ten times faster than MARCO.

# 7 Summary and Future Work

We presented a seed-grow based MSS enumeration algorithm called RIME. Compared to other seed-grow algorithms, RIME can often identify and grow s-seeds without performing any satisfiability checks. Remarkably, on average, RIME performs just 1.18 checks per MSS, and in case of many benchmarks, the number is even lower than 1. Our experimental evaluation showed that RIME is several times faster than contemporary MSS enumeration tools.

In future work, we plan to closely examine and possibly adopt preprocessing [9, 6] and caching [33, 34] techniques that were proposed in the context of MSS/MCS or MUS extraction. Another direction is to examine whether our MSS/MCS enumeration techniques can be somehow applied in the context of MUS enumeration. Finally, we plan to examine if the novel techniques used in RIME can be lifted to other constraint domains, e.g., SMT or LTL, where MSSes, MCSes or MUSes also find an application (see, e.g.,[7, 24, 22, 13, 15]).

# References


[1] M. Fareed Arif, Carlos Mencía, and João Marques-Silva. Efficient axiom pinpointing with EL2MCS. In *KI*, volume 9324 of *Lecture Notes in Computer Science*, pages 225–233. Springer, 2015.

[2] Fahiem Bacchus, Jessica Davies, Maria Tsimpoukelli, and George Katsirelos. Relaxation search: A simple way of managing optional clauses. In *AAAI*, pages 835–841. AAAI Press, 2014.

[3] Fahiem Bacchus and George Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *CAV (2)*, volume 9207 of *LNCS*, pages 70–86. Springer, 2015.

[4] Fahiem Bacchus and George Katsirelos. Finding a collection of MUSes incrementally. In *CPAIOR*, volume 9676 of *LNCS*, pages 35–44. Springer, 2016.

[5] James Bailey and Peter J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *PADL*, pages 174–186. Springer, 2005.

[6] Valeriy Balabanov and Alexander Ivrii. Speeding up MUS extraction with preprocessing and chunking. In *SAT*, volume 9340 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2015.

[7] Jiří Barnat, Petr Bauch, Nikola Beneš, Luboš Brim, Jan Beran, and Tomáš Kratochvíla. Analysing sanity of requirements for avionics systems. *FAoC*, pages 1–19, 2016.

[8] Anton Belov, Marijn Heule, and João Marques-Silva. MUS extraction using clausal proofs. In *SAT*, volume 8561 of *LNCS*, pages 48–57. Springer, 2014.

[9] Anton Belov, Matti Järvisalo, and João Marques-Silva. Formula preprocessing in MUS extraction. In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2013.

[10] Anton Belov and João Marques-Silva. Accelerating MUS extraction with recursive model rotation. In *FMCAD*, pages 37–40. FMCAD Inc., 2011.

[11] Anton Belov and João Marques-Silva. MUSer2: An efficient MUS extractor. *JSAT*, 8:123–128, 2012.

[12] Rachel Ben-Eliyahu and Rina Dechter. On computing minimal models. In *AAAI*, pages 2–8. AAAI Press / The MIT Press, 1993.

[13] Jaroslav Bendík. Consistency checking in requirements analysis. In *ISSTA*, pages 408–411. ACM, 2017.

[14] Jaroslav Bendík, Nikola Beneš, Ivana Černá, and Jiří Barnat. Tunable online MUS/MSS enumeration. In *FSTTCS*, volume 65 of *LIPIcs*, pages 50:1–50:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[15] Jaroslav Bendík and Ivana Černá. Evaluation of domain agnostic approaches for enumeration of minimal unsatisfiable subsets. In *LPAR*, volume 57 of *EPiC Series in Computing*, pages 131–142. EasyChair, 2018.

[16] Jaroslav Bendík and Ivana Černá. MUST: Minimal unsatisfiable subsets enumeration tool. In *TACAS*, pages 135–152. Springer, 2020.

[17] Jaroslav Bendík, Ivana Černá, and Nikola Beneš. Recursive online enumeration of all minimal unsatisfiable subsets. In *ATVA*, volume 11138 of *LNCS*, pages 143–159. Springer, 2018.

[18] Jaroslav Bendík, Elaheh Ghassabani, Michael W. Whalen, and Ivana Černá. Online enumeration of all minimal inductive validity cores. In *SEFM*, volume 10886 of *LNCS*, pages 189–204. Springer, 2018.

[19] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artif. Intell.*, 32(1):97–130, 1987.

[20] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

[21] Alexander Felfernig, Monika Schubert, and Christoph Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. *AI EDAM*, 26(1):53–62, 2012.

[22] Elaheh Ghassabani, Andrew Gacek, Michael W. Whalen, Mats Per Erik Heimdahl, and Lucas G. Wagner. Proof-based coverage metrics for formal verification. In *ASE*, pages 194–199. IEEE Computer Society, 2017.

[23] Éric Grégoire, Jean-Marie Lagniez, and Bertrand Mazure. An experimentally efficient method for (mss, comss) partitioning. In *AAAI*, pages 2666–2673. AAAI Press, 2014.

[24] Ofer Guthmann, Ofer Strichman, and Anna Trostanetski. Minimal unsatisfiable core extraction for SMT. In *FMCAD*, pages 57–64. IEEE, 2016.

[25] Mark H. Liffiton and Ammar Malik. Enumerating infeasibility: Finding multiple MUSes quickly. In *CPAIOR*, volume 7874 of *LNCS*, pages 160–175. Springer, 2013.

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

[27] João Marques-Silva, Federico Heras, Mikolás Janota, Alessandro Previti, and Anton Belov. On computing minimal correction subsets. In *IJCAI*, pages 615–622. IJCAI/AAAI, 2013.

[28] Carlos Mencía, Oliver Kullmann, Alexey Ignatiev, and João Marques-Silva. On computing the union of muses. In *SAT*, volume 11628 of *Lecture Notes in Computer Science*, pages 211–221. Springer, 2019.

[29] Carlos Mencía, Alessandro Previti, and João Marques-Silva. Literal-based MCS extraction. In *IJCAI*, pages 1973–1979. AAAI Press, 2015.

[30] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Accelerated deletion-based extraction of minimal unsatisfiable cores. *JSAT*, 9:27–51, 2014.

[31] Nina Narodytska, Nikolaj Bjørner, Maria-Cristina Marinescu, and Mooly Sagiv. Core-guided minimal correction set and core enumeration. In *IJCAI*, pages 1353–1361. ijcai.org, 2018.

[32] Alessandro Previti and João Marques-Silva. Partial MUS enumeration. In *AAAI*. AAAI Press, 2013.

[33] Alessandro Previti, Carlos Mencía, Matti Järvisalo, and João Marques-Silva. Improving MCS enumeration via caching. In *SAT*, volume 10491 of *Lecture Notes in Computer Science*, pages 184–194. Springer, 2017.

[34] Alessandro Previti, Carlos Mencía, Matti Järvisalo, and João Marques-Silva. Premise set caching for enumerating minimal correction subsets. In *AAAI*, pages 6633–6640. AAAI Press, 2018.

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

[36] Emanuel Sperner. Ein satz über untermengen einer endlichen menge. *Mathematische Zeitschrift*, 27(1):544–548, 1928.

[37] Roni Tzvi Stern, Meir Kalech, Alexander Feldman, and Gregory M. Provan. Exploring the duality in conflict-directed model-based diagnosis. In *AAAI*. AAAI Press, 2012.