Replication-Guided Enumeration of Minimal Unsatisfiable Subsets

Jaroslav Bendík(✉) and Ivana Černá

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

Abstract. In many areas of computer science, we are given an unsatisfiable Boolean formula F in CNF, i.e. a set of clauses, with the goal to identify minimal unsatisfiable subsets (MUSes) of F. The more MUSes are identified, the better insight into F's unsatisfiability is obtained. Unfortunately, finding even a single MUS can be very time consuming since it naturally subsumes repeatedly solving the satisfiability problem, and thus a complete MUS enumeration is often practically intractable. Therefore, contemporary MUS enumeration algorithms tend to identify as many MUSes as possible within a given time limit. In this work, we present a novel MUS enumeration algorithm. Compared to existing algorithms, our algorithm is much more frugal in the number of performed satisfiability checks. Consequently, our algorithm is often able to find substantially more MUSes than contemporary algorithms.

1 Introduction

Given an unsatisfiable set $F = \{c_1, \dots, c_n\}$ of Boolean clauses, a minimal unsatisfiable subset (MUS) of F is a set $M \subseteq F$ such that M is unsatisfiable and for all $c \in M$ the set $M \setminus \{c\}$ is satisfiable. MUSes represent the minimal reasons for F's unsatisfiability, and as such, they find applications in a wide variety of domains including, e.g., formal equivalence checking [18], Boolean function bi-decomposition [17], counter-example guided abstraction refinement [1], circuit error diagnosis [21], type debugging in Haskell [37], and many others [2,23–25,32].

The more MUSes are identified, the better insight into the unsatisfiability of F is obtained. However, there can be, in general, up to exponentially many MUSes w.r.t. $|F|$, and thus, the complete MUS enumeration is often practically intractable. Consequently, there have been proposed several algorithms, e.g., [4,5,9,12,27,30,33,36], that enumerate MUSes *online*, i.e., one by one, and attempt to identify as many MUSes as possibly within a given time limit.

Many of the algorithms can be classified as *seed-shrink* algorithms [11]. A seed-shrink algorithm gradually explores subsets of F; *explored* subsets are those,

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).

© Springer Nature Switzerland AG 2020
H. Simonis (Ed.): CP 2020, LNCS 12333, pp. 37–54, 2020.
https://doi.org/10.1007/978-3-030-58475-7_3

whose satisfiability is already known by the algorithm, and *unexplored* are the others. To find each single MUS, a seed-shrink algorithm first identifies an unsatisfiable unexplored subset, called a *u-seed*, and then *shrinks* the u-seed into a MUS via a single MUS extraction subroutine. The exact way of finding and shrinking u-seeds differ for individual algorithms. In general, the algorithms find a u-seed by repeatedly checking unexplored subsets for satisfiability, via a SAT solver, until they find an unsatisfiable one. Naturally, the performance of the algorithms highly depends on the number these satisfiability checks.

In this paper, we propose a novel seed-shrink algorithm called UNIMUS. The algorithm employs two novel techniques for finding u-seeds. One of the techniques works on the same principle as the existing seed-shrinks algorithms do: it checks unexplored subsets for satisfiability until it finds a u-seed. The novelty is in the selection of subsets to be checked; we use the union of already explored MUSes to identify a search-space where we can quickly find new u-seeds. The other technique for finding u-seeds works on a fundamentally different principle; we cheaply (in polynomial time) *deduce* that some unexplored subsets are unsatisfiable. We experimentally compare UNIMUS with 4 contemporary MUS enumeration algorithms on a standard collection of benchmarks. UNIMUS outperforms all of its competitors on majority of the benchmarks. Remarkably, UNIMUS often finds 10–100 times more MUSes than its competitors.

2 Preliminaries

A Boolean formula $F = \{c_1, \dots, c_n\}$ in a conjunctive normal form is a set of clauses over a set of variables $\mathit{Vars}(F)$. A clause $c = \{l_1, \dots, l_k\}$ is a set of literals. A literal is either a variable $x \in \mathit{Vars}(F)$ or its negation $\neg x$. A truth assignment I is a mapping $\mathit{Vars}(F) \rightarrow \{\top, \bot\}$. A clause $c \in F$ is satisfied by an assignment I iff $I(x) = \top$ for some $x \in c$ or $I(y) = \bot$ for some $\neg y \in c$. The formula F is satisfied by I iff I satisfies every clause $c \in F$; in such a case I is a *model* of F. Finally, F is *satisfiable* if it has a model; otherwise F is *unsatisfiable*. Hereafter, we use F to denote the input formula of interest, capital letters, e.g. N, T, K to denote subsets of F, small letters, e.g., c, d, c_i to denote clauses of F, and small letters, e.g., x, y, x_i to denote variables of F. Given a set X, we use $|X|$ to denote the cardinality of X, and $\mathcal{P}(X)$ to denote the power-set of X.

2.1 Minimum Unsatisfiability

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

Note that the minimality refers to a set minimality, not to minimum cardinality. Therefore, there can be MUSes with different cardinalities and in general, there can be up to exponentially many MUSes of F w.r.t. $|F|$ (see [35]). We write $\mathtt{UMUS}_F$ to denote the union of all MUSes of F.

Example 1. Assume that we are given a formula $F = \{c_1 = \{x_1\}, c_2 = \{\neg x_1\}, c_3 = \{x_2\}, c_4 = \{\neg x_1, \neg x_2\}\}$. There are two MUSes: $\{c_1, c_2\}$ and $\{c_1, c_3, c_4\}$, and $\texttt{UMUS}_F = F$. The power-set of F is illustrated in Fig. 1a.

Definition 2 (critical clause). *Let U be an unsatisfiable subset of F. A clause $c \in U$ is* critical *for U iff $U \setminus \{c\}$ is satisfiable.*

Note that if c is critical for U then c has to be contained in every unsatisfiable subset of U, and especially in every MUS of U. Furthermore, note that U is a MUS if and only if every clause $c \in U$ is critical for U.

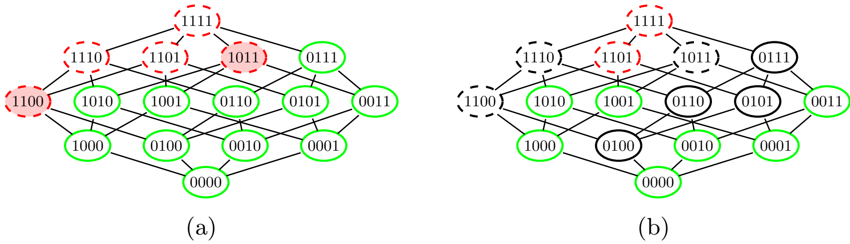

Fig. 1. a) illustrates the power-set of the set F of clauses from Example 1. We encode individual subsets of F as bit-vectors; for example, the subset $\{c_1, c_3, c_4\}$ is written as 1011. The subsets with a dashed border are the unsatisfiable subsets, and the others are satisfiable subsets. The MUSes are filled with a background color. b) illustrates the explored and unexplored subsets from Example 2. (Color figure online)

Finally, we exploit capabilities of contemporary SAT solvers. Given a set $N \subseteq F$, many of SAT solvers are able to provide an *unsat core* of N when N is unsatisfiable, and a *model* I of N when N is satisfiable. The unsat core is often small, yet not necessarily minimal, unsatisfiable subset of N. The model I, on the other hand, induces the *model extension* E of N defined as $E = \{c \in F \mid I$ satisfies $c\}$. Note that $N \subseteq E$ and E is satisfiable (I is its model).

2.2 Unexplored Subsets

Every online MUS enumeration algorithm during its computation gradually *explores* satisfiability of individual subsets of F. *Explored* subsets are those whose satisfiability is already known by the algorithm and *unexplored* are the other ones. We write `Unexplored` to denote the set of all unexplored subsets. Furthermore, we classify the unexplored subsets either as *s-seeds* or *u-seeds*; *s-seeds* are satisfiable unexplored subsets and *u-seeds* are unsatisfiable unexplored subsets.

Note that if a subset U of F is unsatisfiable, then also every superset of U is unsatisfiable. Therefore, when U becomes explored, then also all supersets of U become explored. Dually, when a satisfiable S, $S \subseteq F$, becomes explored, then also all subsets of S become explored since they are necessarily also satisfiable.

Algorithm 1: Seed-Shrink Scheme

```
1 Unexplored ← P(F)
2 while there is a u-seed do
3     S ← find a u-seed
4     S_mus ← shrink(S)
5     output S_mus
6     Unexplored ← Unexplored \ {X | X ⊆ S_mus ∨ X ⊇ S_mus}
```

Observation 1. *If N is a u-seed, then every unsatisfiable subset of N is also a u-seed. Dually, if N is an s-seed then every satisfiable superset of N is an s-seed.*

Example 2 Figure 1b shows a possible state of exploration of the power-set of F from Example 1. There are seven explored satisfiable subsets (green with solid border), two explored unsatisfiable subsets (red with dashed border), four s-seeds (black with solid border), and three u-seeds (black with dashed border).

Note that based on `Unexplored`, we can *mine* some critical clauses for some u-seeds. For instance, in Example 2, we can see that c_2 is critical for the u-seed $U = \{c_1, c_2, c_3\}$ since $U \setminus \{c_2\}$ is explored and thus satisfiable (Observation 1).

Definition 3 (minable critical). *Let U be a u-seed and c a critical clause for U. The clause c is a* minable critical clause *for U if $U \setminus \{c\} \notin$ `Unexplored`.*

Details on how exactly we represent and perform operations over `Unexplored` are postponed to Sect. 3.5.

2.3 Seed-Shrink Scheme

Many of existing MUS enumeration algorithms, e.g., [6,9,12,28,36], and including the algorithm we present in this paper, can be classified as *seed-shrink* algorithms [11]. The base scheme (Algorithm 1) works iteratively. Each iteration starts by identifying a u-seed S. Then, the u-seed S is *shrunk* into a MUS S_{mus} via a single MUS extraction subroutine. The iteration is concluded by removing all subsets and all supersets of the MUS from `Unexplored` since none of them can be another MUS. The computation terminates once there is no more u-seed.

The exact way S is found differs for individual seed-shrink algorithms. In general, existing algorithms identify S by repeatedly picking and checking an unexplored subset for satisfiability until they find a u-seed. The algorithms vary in *which* and *how* many unexplored subsets they check. In general, it is worth to minimize the number of these checks as they are very expensive. Also, it generally holds that the closer (w.r.t. set containment) the u-seed is to a MUS, the easier it is to shrink the u-seed into a MUS. As for the shrinking, all the algorithms can collect and exploit all the minable critical clauses for S; these clauses have to be contained in every MUS of S and thus their prior knowledge can significantly speed up the MUS extraction. However, the exact way the algorithms find the MUS differ for individual algorithms. See Sects. 3.4 and 4 for more details.

Algorithm 2: UNIMUS

1 `Unexplored` $\leftarrow \mathcal{P}(F)$; $B \leftarrow \emptyset$
2 **while** `Unexplored` $\neq \emptyset$ **do**
3 $\quad B \leftarrow$ refine(B)
4 $\quad$ UNIMUSCore(B)

3 Algorithm

Our MUS enumeration algorithm, called UNIMUS, is based on the seed-shrink scheme. It employs a novel shrinking procedure. Moreover, it employs two novel approaches for finding u-seeds. One of the approaches is based on the same idea as the contemporary seed-shrink algorithms: to find a u-seed, we are repeatedly picking and checking for satisfiability (via a SAT solver) an unexplored subset until we identify a u-seed. The novelty is in the choice of unexplored subsets to be checked. Briefly, we maintain a *base* B and a *search-space* $\texttt{Unex}_B = \{X \mid X \in \texttt{Unexplored} \wedge X \subseteq B\}$ that is induced by the base. UNIMUS searches for u-seeds only within $\texttt{Unex}_B$. The base B (and thus $\texttt{Unex}_B$) is maintained in a way that allows identifying u-seeds with performing only few satisfiability checks. Moreover, the u-seeds are very close to MUSes and thus relatively easy to shrink.

Our other approach for finding u-seeds is based on a fundamentally different principle. Instead of checking unexplored subsets for satisfiablity via a SAT solver, we *deduce* that some unexplored subsets are unsatisfiable. The deduction is based on already identified MUSes and it is very cheap (polynomial).

3.1 Main Procedure

UNIMUS (Algorithm 2) first initializes `Unexplored` to $\mathcal{P}(F)$ and the base B to $\emptyset$. Then, it in a while-loop repeats two procedures: refine that updates the base B, and UNIMUSCore that identifies MUSes in the search-space $\texttt{Unex}_B = \{X \mid X \in \texttt{Unexplored} \wedge X \subseteq B\}$. The algorithm terminates once $\texttt{Unexplored} = \emptyset$.

UNIMUSCore (Algorithm 3) works iteratively. Each iteration starts by picking a *maximal element* N of $\texttt{Unex}_B$, i.e., $N \in \texttt{Unex}_B$ such that $N \cup \{c\} \notin \texttt{Unex}_B$ for every $c \in B \setminus N$. Subsequently, a procedure isSAT is used to determine, via a SAT solver, the satisfiability of N. Moreover, in dependence of N's satisfiability, isSAT returns either an unsat core K or a model extension E of N. If N is unsatisfiable, the algorithm shrinks the core K into a MUS K_{mus} and removes from `Unexplored` all subsets and all supersets of K_{mus}. Subsequently, a procedure replicate is invoked which attempts to identify additional MUSes in $\texttt{Unex}_B$.

In the other case, when N is satisfiable, we remove all subsets of E from `Unexplored`. Then, N is used to guide the algorithm into a search-space with more minable critical clauses. In particular, we know that for every $c \in B \setminus N$ the set $N \cup \{c\}$ is explored and thus unsatisfiable (Observation 1). Consequently, every clause $c \in B \setminus N$ is critical for $N \cup \{c\}$ and especially for every u-seed contained in $N \cup \{c\}$. Moreover, all these critical clauses are minable critical as

Algorithm 3: UNIMUSCore(B)

```
1  while {N ∈ Unexplored | N ⊆ B} ≠ ∅ do
2      N ← a maximal element of {X | X ∈ Unexplored ∧ X ⊆ B}          // Unex_B
3      (sat?, E, K) ← isSAT(N)
4      if not sat? then
5          K_mus ← shrink(K)
6          output K_mus
7          Unexplored ← Unexplored \ {X | X ⊆ K_mus ∨ X ⊇ K_mus}
8          replicate(K_mus, N)
9      else
10         Unexplored ← Unexplored \ {X | X ⊆ E}
11         if |B \ N| > 1 then
12             for c ∈ B \ N do  UNIMUSCore(N ∪ {c})
```

all subsets of N were removed from Unexplored. If $N \cup \{c\} = B$ then c is minable critical for every u-seed in the current search-space. Otherwise, if $|B \setminus N| > 1$, we recursively call UNIMUSCore with a base $B' = N \cup \{c\}$ for every $c \in B \setminus N$.

The procedures refine, shrink, and replicate are described in Sects. 3.2, 3.4, and 3.3, respectively. All procedures of UNIMUS follow two rules about Unexplored. First, Unexplored is *global*, i.e., shared by the procedures. Second, we remove an unsatisfiable set from Unexplored only if the set is a superset of an explicitly identified MUS. Consequently, no MUS can be removed from Unexplored without being explicitly identified. Thus, when Algorithm 2 terminates (i.e., Unexplored $= \emptyset$), it is guaranteed that all MUSes were identified.

Heuristics. According to the above description, UNIMUSCore terminates once all subsets, and especially all MUSes, of B become explored. However, based on our empirical experience, UNIMUSCore can get into a situation such that there is a lot of s-seeds in $\mathtt{Unex}_B$ but only few or even no u-seed. Consequently, the MUS enumeration can get stuck for a while. To prevent such a situation, we track the number of subsequent iterations of UNIMUSCore in which the set N was satisfiable. If there are 5 such subsequent iterations, we backtrack from the current recursive call of UNIMUSCore.

3.2 The Base and the Search-Space

The base B is modified in two situations. The first situation is the case of recursive calls of UNIMUSCore which was described in the previous section. Here, we describe the second situation which is an execution of the procedure refine before each top-level call of UNIMUSCore. The goal is to identify a base B such that u-seeds in $\mathtt{Unex}_B$ can be easily found and are relatively easy to shrink.

We exploit the union $\mathtt{UMUS}_F$ of all MUSes of F. Assume that we set B to $\mathtt{UMUS}_F$. Since every MUS of F is contained in $\mathtt{UMUS}_F$, the induced search-space

Algorithm 4: refine(B)

```
1  while Unexplored ≠ ∅ do
2      T ← a maximal element of Unexplored
3      (sat?, E, K) ← isSAT(T)
4      if not sat? then
5          K_mus ← shrink(K)
6          output K_mus
7          Unexplored ← Unexplored \ {X | X ⊆ K_mus ∨ X ⊇ K_mus}
8          return B ∪ K_mus
9      else Unexplored ← Unexplored \ {X | X ⊆ T}
10 return B
```

$\texttt{Unex}_B$ would contain all MUSes. Moreover, compared to the whole F, the cardinality of $\texttt{UMUS}_F$ can be relatively small and thus the u-seeds in $\texttt{Unex}_B$ would be easy to shrink. Unfortunately, based on recent studies [14,31], computing $\texttt{UMUS}_F$ is often practically intractable even for small input formulas. Thus, instead of initially computing $\texttt{UMUS}_F$, we use as the base B just an under-approximation of $\texttt{UMUS}_F$. Initially, we set B to $\emptyset$ (Algorithm 2, line 1) and in each call of refine we attempt to refine (enlarge) the under-approximation. Eventually, B becomes $\texttt{UMUS}_F$ and, thus, eventually the search-space will contain all MUSes of F.

The procedure refine (Algorithm 4) attempts to enlarge B with an unexplored MUS. In each iteration, it picks a maximal element T of Unexplored, i.e., $T \in$ Unexplored such that $T \cup \{c\} \notin$ Unexplored for every $c \in F \setminus T$. Then, T is checked for satisfiability via the procedure isSAT. If T is unsatisfiable, then isSAT also returns an unsat core K of T, refine shrinks the core K into a MUS K_{mus} and based on K_{mus} updates the set Unexplored. Subsequently, refine terminates and returns an updated base $B' = B \cup \{K_{mus}\}$. Otherwise, if T is satisfiable, refine removes all subsets of T from Unexplored and continues with a next iteration.

There is no guarantee that each call of refine indeed enlarges B. One possibility is the corner case when all MUSes are already explored, but there are some s-seeds left. Another possibility is that the search-space $\texttt{Unex}_B$ was not completely explored in the last call of UNIMUSCore due to the preemptive termination heuristic. Thus, refine might identify a MUS that is a subset of B. Also, note that the procedure refine is very similar to a MUS enumeration algorithm MARCO [28]. The difference is that refine finds only a single unexplored MUS whereas MARCO finds them all (see Sect. 4 for details).

3.3 MUS Replication

We now describe the procedure replicate(K_{mus}, N) that based on an identified MUS K_{mus} of N attempts to identify additional unexplored MUSes. The procedure follows the seed-shrink scheme, i.e., it searches for u-seeds and shrinks them to MUSes. However, contrary to existing seed-shrink algorithms which identify

Algorithm 5: replicate(K_{mus}, N)

```
1  𝓜 ← {K_mus}; rStack ← ⟨K_mus⟩
2  while rStack is not empty do
3      M ← rStack.pop()
4      for c ∈ M do
5          if c is minable critical for N then continue
6          S ← propagate(M, c, N, 𝓜)
7          if S is null then continue
8          S_mus ← shrink(S)
9          output S_mus
10         Unexplored ← Unexplored \ {X | X ⊆ S_mus ∨ X ⊇ S_mus}
11         𝓜 ← 𝓜 ∪ {S_mus}
12         rStack.push(S_mus)
```

u-seeds via a SAT solver, replicate identifies u-seeds with a cheap (polynomial) deduction technique; we call the technique *MUS replication*.

Each call of replicate possibly identifies several unexplored MUSes and all these MUSes are subsets of N. Note that since N is a subset of the base B in Algorithm 3, all MUSes identified by replicate are contained in the search-space $\texttt{Unex}_B$. Also, note that when replicate is called, K_{mus} is the only explored MUS of N (since N was a u-seed that we shrunk to K_{mus}). In the following, we will use $\mathcal{M}$ to denote the set of all explored MUSes of N, i.e., initially, $\mathcal{M} = \{K_{mus}\}$.

Main Procedure. The main procedure of replicate (Algorithm 5) maintains two data-structures: the set $\mathcal{M}$ and a stack *rStack*. The computation starts by initializing both $\mathcal{M}$ and *rStack* to contain the MUS K_{mus}. The rest of replicate is formed by two nested loops. In each iteration of the outer loop, replicate pops a MUS M from the stack. In the nested loop, M is used to identify possibly several unexplored MUSes. In particular, for each clause $c \in M$ the algorithm attempts to identify a u-seed S such that $M \setminus \{c\} \subset S \subseteq N \setminus \{c\}$. Observe that if c is minable critical for N then such a u-seed cannot exist; thus, we skip such clauses. The attempt to find such S is carried out by a procedure propagate. If propagate fails to find the u-seed, the inner loop proceeds with a next iteration. Otherwise, the u-seed S is shrunk into a MUS S_{mus} and the set Unexplored is appropriately updated. The iteration is concluded by adding S_{mus} to $\mathcal{M}$ and also pushing S_{mus} to *rStack*, i.e., each identified MUS is used to possibly identify additional MUSes. The computation terminates once *rStack* becomes empty.

Propagate. The procedure propagate is based on the well-known concepts of *backbone literals* and *unit propagation* [16,26]. Given a formula P, a literal l is a *backbone literal* of P iff every model of P satisfies $\{l\}$. A *backbone* of P is a set of backbone literals of P. If A is a backbone of P then A is also a backbone of every superset of P. A clause d is a *unit clause* iff $|d| = 1$. Note that if d is a unit clause of P then the literal $l \in d$ is a backbone literal of P.

Given a backbone A of P, the *backbone propagation* can simplify P and possibly show that P is unsatisfiable. In particular, for every $l \in A$ and every clause $d \in P$ such that $\neg l \in d$, we remove the literal $\neg l$ from d (since no model of P can satisfy $\neg l$). If a new unit clause emerges during the propagation, the backbone literal that forms the unit clause will be also propagated. If the propagation reduces a clause to an empty clause, then the original P is unsatisfiable.

The procedure `propagate` employs backbone propagation to identify a u-seed S such that $M \setminus \{c\} \subset S \subseteq N \setminus \{c\}$. Observe that since M is unsatisfiable and $M \setminus \{c\}$ is satisfiable, then the set $A = \{\neg l \,|\, l \in c\}$ is a backbone of every such S. Thus, one can pick such S and attempt to show, via propagating A, that S is unsatisfiable. However, there are too many such S to choose from. Moreover, we need to guarantee that we find S that is both unsatisfiable and unexplored. Thus, instead of fixing a particular S and then trying to show its unsatisfiability, we attempt to gradually build such S. Initially, we set S to $M \setminus \{c\}$ and we step-by-step add clauses from $N \setminus \{c\}$ to S. The addition of the clauses is driven by a currently known backbone A of S and also by the set $\mathcal{M}$ of explored MUSes to ensure that the resulting S is a u-seed.

Observation 2. *For every* ***unsatisfiable*** *S, $S \subseteq N \setminus \{c\}$, it holds that $S \in$* `Unexplored` *(i.e., S is a u-seed) if and only if $\forall_{X \in \mathcal{M}} S \not\supseteq X$.*

Proof. In UNIMUS, we remove from `Unexplored` only unsatisfiable sets that are supersets of explored MUSes and all explored MUSes of N are stored in $\mathcal{M}$.

Observation 2 shows which clauses *can be added* to the initial S while ensuring that if we finally obtain an unsatisfiable S, then the final S will be unexplored. Note that the initial $S = M \setminus \{c\}$ trivially satisfies $\forall_{X \in \mathcal{M}} M \setminus \{c\} \not\supseteq X$ since it is satisfiable (M is a MUS). In the following, we show which clauses *should be added* to S to eventually make it unsatisfiable.

Definition 4 (operation $\setminus\setminus$). *Let d be a clause and A be a set of literals. The the binary operation $d \setminus\setminus A$ creates the clause $d \setminus\setminus A = \{l \mid l \in d \text{ and } \neg l \notin A\}$.*

Definition 5 (units, violated). *Let S be a set such that $M \setminus \{c\} \subset S \subseteq N \setminus \{c\}$, and let A be a backbone of S. We define the following sets:*

$$units(S, A) = \{d \in N \setminus \{c\} \mid \forall_{X \in \mathcal{M}} S \cup \{d\} \not\supseteq X \wedge |d \setminus\setminus A| = 1\}$$
$$violated(S, A) = \{d \in N \setminus \{c\} \mid \forall_{X \in \mathcal{M}} S \cup \{d\} \not\supseteq X \wedge |d \setminus\setminus A| = 0\}$$

Informally, a clause $d \in N \setminus \{c\}$ belongs to $units(S, A)$ ($violated(S, A)$) if the propagation of A would simplify d to a unit clause (empty clause) and, simultaneously, $d \in S$ or d can be added to S in a harmony with Observation 2.

Observation 3. *For every S such that $M \setminus \{c\} \subset S \subseteq N \setminus \{c\}$, a backbone A of S, and a clause $d \in units(S, A)$, it holds that $A \cup d \setminus\setminus A$ is a backbone of $S \cup \{d\}$.*

Proof. Assume that $d = \{l, l_0, \ldots, l_k\}$ where $l = d \setminus\setminus A$ and $\{\neg l_0, \ldots, \neg l_k\} \subseteq A$. Since A is a backbone of S then every model of S satisfies $\{\{\neg l_0\}, \ldots, \{\neg l_k\}\}$. Consequently, every model of $S \cup \{d\}$ satisfies $\{l\}$.

Algorithm 6: propagate(M,c,N,$\mathcal{M}$)

```
1 S ← M \ {c}; A ← {¬l | l ∈ c}; H ← {c}
2 while units(S, A) \ H ≠ ∅ ∧ violated(S, A) = ∅ do
3     d ← choose d ∈ units(S, A) \ H
4     S ← S ∪ {d}; H ← H ∪ {d}; A ← A ∪ d \\ A
5 if violated(S, A) = ∅ then return null
6 else
7     d ← choose d ∈ violated(S, A)
8     return S ∪ {d}
```

Observation 4. *For every S such that $M \setminus \{c\} \subset S \subseteq N \setminus \{c\}$, a backbone A of S, and a clause $d \in violated(S, A)$, it holds that $S \cup \{d\}$ is unsatisfiable.*

Proof. Assume that $d = \{l_0, \ldots, l_q\}$. As $d \in violated(S, A)$ then $\{\neg l_0, \ldots, \neg l_q\} \subseteq A$. Since A is a backbone of S then every model of S satisfies $\{\{\neg l_0\}, \ldots, \{\neg l_q\}\}$, i.e., no model of S satisfies d.

The procedure propagate (Algorithm 6) maintains three data structures: the sets S and A, and an auxiliary set H for storing clauses that were used to enlarge A. Initially, $S = M \setminus \{c\}$, $A = \{\neg l \,|\, l \in c\}$ and $H = \{c\}$. In each iteration, propagate picks a clause $d \in units(S, A) \setminus H$ and, based on Observation 3, adds d to S and to H, and the literal of $d \setminus\setminus A$ to A. The loop terminates once there is no more backbone literal to propagate ($units(S, A) \setminus H = \emptyset$), or once $violated(S, A) \neq \emptyset$. If $violated(S, A) = \emptyset$, propagate failed to find a u-seed. Otherwise, propagate picks a clause $d \in violated(S, A)$ and returns the u-seed $S \cup \{d\}$.

Finally, note that backbone propagation is cheap (polynomial) but it is not a *complete* technique for deciding satisfiability. Consequently, it can happen that there is a u-seed S, $M \setminus \{c\} \subset S \subseteq N \setminus \{c\}$, but MUS replication fails to find it.

3.4 Shrink

Existing seed-shrink algorithms can be divided into two groups. Algorithms from one group, e.g. [4,5,13], implement the shrinking via a custom single MUS extractor that fully shares information with the overall MUS enumeration process. Consequently, all information obtained during shrinking can be exploited by the overall MUS enumeration algorithm and vice versa. Algorithms from the other group, e.g. [9,12,28], implement the shrinking via an external, black-box, single MUS extraction tool. The advantage is that one can always use the currently best available single MUS extractor to implement the shrinking. On the other hand, the black-box extractor cannot fully share information with the overall MUS enumeration algorithm. The only output of the extractor is a MUS of a given u-seed N. As for the input, besides the u-seed N, contemporary single MUS extractors, e.g., [4,8], allow the user to provide also a set C of clauses that are critical for N since a prior knowledge of C can significantly speed up the

```
Algorithm 7: criticalExtension(N)
1  C ← collect all minable critical clauses for N
2  Q ← C
3  while Q ≠ ∅ do
4  |   c ← pick c ∈ Q
5  |   Q ← Q \ {c}
6  |   for l ∈ c do
7  |   |   M ← {d ∈ N | ¬l ∈ d}
8  |   |   if |M| = 1 and M ∩ C = ∅ then
9  |   |   |   C ← C ∪ M; Q ← Q ∪ M
10 return C
```

extraction. Thus, contemporary algorithms [9,12,28] collect all minable critical clauses for N and pass them to the single MUS extractor together with N.

In our work, we follow the black-box approach, i.e., to find a MUS of a u-seed N, we first identify a set C of clauses that are critical for N and then pass N and C to an external single MUS extractor (e.g., [4,8]). However, contrary to existing algorithms, we identify more than just *minable* critical clauses for N. We introduce a technique that, based on the minable critical clauses for N, can cheaply *deduce* that some other clauses are critical for N. We call the deduction technique *critical extension* and it is based on the following observation.

Observation 5. *Let N be a u-seed, $c \in N$ a critical clause for N, and $l \in c$. Moreover, let $M \subseteq N$ be the set of all clauses of N that contain the literal $\neg l$. If $|M| = 1$ then the clause $d \in M$ is critical for N, i.e. $N \setminus \{d\}$ is satisfiable.*

Proof. Assume that $N \setminus \{d\}$ is unsatisfiable. Since c is critical for N, then c is critical also for $N \setminus \{d\}$. Thus, $N \setminus \{c, d\}$ has a model and every its model satisfies $\{\neg l\}$ (as $l \in c$) which contradicts that $\neg l$ is contained only in d.

The critical extension technique (Algorithm 7) takes as an input a u-seed N and outputs a set C of clauses that are critical for N. The algorithm starts by collecting (see Sect. 3.5) all minable critical clauses for N and stores them to C and also to an auxiliary set Q. The rest of the computation works iteratively. In each iteration, the algorithm picks and removes a clause c from Q and employs Observation 5 on c. In particular, for each literal $l \in c$, the algorithm builds the set $M = \{d \in N \mid \neg l \in d\}$. If M contains only a single clause, say d, and $d \notin C$, then d is a new critical clause for N and thus it is added to C and to Q. The computation terminates once Q becomes empty.

Our technique is similar to *model rotation* [4,7] which identifies additional critical clauses based on a critical clause c of N and a model of $N \setminus \{c\}$. The difference is that we do not need the model. Another approach [38] that also does not need the model is based on *rotation edges* in a *flip graph* of F.

3.5 Representation of Unexplored Subsets

To maintain the set `Unexplored`, we adopt a representation that was originally proposed by Liffiton et al. [28] and nowadays is used by many MUS enumeration algorithms (e.g., [5,9,12,33]). Given a formula $F = \{c_1, \ldots, c_n\}$, we introduce a set $X = \{x_1, \ldots, x_n\}$ of Boolean variables. Note that every valuation of X corresponds to a subset of F and vice versa. To represent `Unexplored`, we maintain two formulas, map^+ and map^-, over X such that every model of $map^+ \wedge map^-$ corresponds to an element of `Unexplored` and vice versa. In particular:

- Initially, $\texttt{Unexplored} = \mathcal{P}(F)$, thus we set $map^+ = map^- = \top$.
- To remove a set U, $U \subseteq F$, and all supersets of U from `Unexplored`, we add to map^- the clause $\bigvee_{c_i \in U} \neg x_i$.
- Dually, to remove a set S, $S \subseteq F$, and all subsets of S from `Unexplored`, we add to map^+ the clause $\bigvee_{c_i \notin S} x_i$.

To get an arbitrary element of `Unexplored`, one can ask a SAT solver for a model of $map^+ \wedge map^-$. However, in UNIMUS, we need to obtain more specific unexplored subsets. Given a set B, we require a *maximal* element N of $\texttt{Unex}_B = \{X \mid X \in \texttt{Unexplored} \wedge X \subseteq B\}$. One of SAT solvers that allows us to obtain such N is miniSAT [20]. To obtain N, we instruct miniSAT to fix the values of the variables $\{x_i \mid c_i \notin B\}$ to $\perp$ and ask it for a maximal model of $map^+ \wedge map^-$.

Finally, given a u-seed U, to collect all minable critical clauses of U we check for each $c \in U$ whether $U \setminus \{c\}$ corresponds to a model of $map^+ \wedge map^-$. To do it efficiently, observe that the information represented by map^- is irrelevant. Intuitively, map^- *requires an absence* of clauses, and since U satisfies map^- (it is a u-seed), the set $U \setminus \{c\}$ also satisfies map^-. Thus, c is minable critical for U iff $U \setminus \{c\}$ does not correspond to a model of map^+.

4 Related Work

MUS enumeration was extensively studied in the past decades and many various algorithms were proposed (see e.g., [4–6,9,10,12,19,21,22,27,29,33,34,36]). In the following, we briefly describe contemporary online MUS enumeration algorithms.

FLINT [33] computes MUSes in *rounds* and each round consists of two phases: *relaxing* and *strengthening*. In the relaxing phase, the algorithm starts with an unsatisfiable formula U and weakens it by iteratively relaxing its unsat core until it gets a satisfiable formula S. The intermediate unsat cores are used to extract MUSes. The resulting satisfiable formula S is passed to the second phase, where the formula is again strengthened to an unsatisfiable formula that is used in the next round as an input for the relaxing phase.

MARCO [28] and ReMUS [12] are algorithms based on the seed-shrink scheme. That is, similarly as UNIMUS, to find each single MUS, the algorithms first identify a u-seed and then shrink the u-seed into a MUS. Before the shrinking, the algorithms first reduce the u-seed to its unsat core (provided by a SAT solver)

and they also collect the minable critical clauses for the u-seed. The shrinking is performed via an external, black-box subroutine. The main difference between the two algorithms is how they find the u-seed. MARCO is iteratively picking and checking for satisfiability a maximal unexplored subset of F, until it finds a u-seed S. Since unsatisfiable subsets of F are naturally more concentrated among the larger subsets, MARCO usually performs only few checks to find the u-seed. However, large u-seeds are generally hard to shrink. Thus, the efficiency of MARCO crucially depends on the capability of the SAT solver to provide a reasonably small unsat core of S. ReMUS, in contrast to MARCO, tends to identify u-seeds that are relatively small and thus easy to shrink. In particular, the initial u-seed S is found among the maximal unexplored subsets of F and then shrunk into a MUS S_{mus}. To find another MUS, ReMUS picks some R such that $S_{mus} \subset R \subset S$, and recursively searches for u-seeds among the maximal unexplored subsets of R. The (expected) size of the u-seeds thus decreases with each recursive call. The disadvantage of ReMUS is that it was designed as a domain-agnostic MUS enumeration algorithm, i.e., F can be a set of constraints in an arbitrary logic (e.g., SMT or LTL). Consequently, ReMUS does not directly employ any techniques that are specific for the SAT (Boolean) domain (such as the MUS replication and the critical extension that we use in UNIMUS).

MCSMUS [5] can be seen as another instantiation of the seed-shrink scheme. Contrary to MARCO, ReMUS, and UNIMUS, MCSMUS implements the shrinking via a custom single MUS extraction procedure that fully shares information and works in a synergy with the overall MUS enumeration algorithm. For example, satisfiable subsets of F that are identified during shrinking are remembered by MCSMUS and exploited in the further computation.

5 Experimental Evaluation

We have experimentally compared UNIMUS with four contemporary MUS enumeration algorithms: MARCO [28], MCSMUS [5], FLINT [33], and ReMUS [12]. A precompiled binary of FLINT was kindly provided to us by its author, Nina Narodytska. The other three tools are available at https://sun.iwu.edu/~mliffito/marco/, https://bitbucket.org/gkatsi/mcsmus/src, and https://github.com/jar-ben/mustool. The implementation of UNIMUS is available at: https://github.com/jar-ben/unimus.

We used the best (default) settings for all evaluated tools. Note that individual tools use different SAT solvers and different shrinking subroutines. ReMUS, MARCO and FLINT use the tool muser2 [8] for shrinking, MCSMUS uses its custom shrinking subroutine, and in UNIMUS we employ the shrinking procedure from the MCSMUS tool. As for SAT solvers, MARCO and ReMUS use miniSAT [20], MCSMUS uses glucose [3] and UNIMUS uses CaDiCaL [15].

As benchmarks, we used a collection of 291 CNF formulas from the MUS track of the SAT 2011 Competition.[1] This collection is standardly used in MUS related papers, including the papers that presented our four competitors. All

[1] http://www.cril.univ-artois.fr/SAT11/.

experiments were run using a time limit of 3600 s and computed on an AMD 16-Core Processor and 1 TB memory machine running Debian Linux. Complete results are available in an online appendix: https://www.fi.muni.cz/~xbendik/research/unimus.

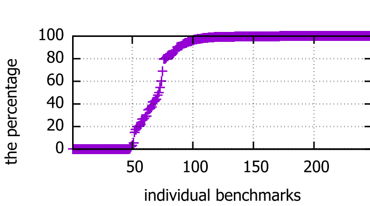

Fig. 2. Percentage of MUSes found by MUS replication.

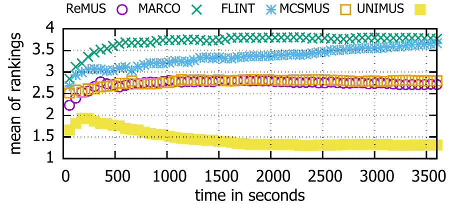

Fig. 3. 5% truncated mean of rankings after each 60 s.

Manifestation of MUS Replication. MUS replication is a crucial part of UNIMUS as, to the best of our knowledge, it is the first existing technique that identifies u-seeds in polynomial time. Therefore, we are interested in what is the percentage of u-seeds, and thus MUSes, that UNIMUS identifies via MUS replication. Figure 2 shows this percentage (y-axis) for individual benchmarks (x-axis); the benchmarks are sorted by the percentage. We computed the percentage only for the 248 benchmarks where UNIMUS found at least 5 MUSes. Remarkably, in case of 161 benchmarks, the percentage is higher than 90%, and in case of 130 benchmarks, it is higher than 99%.[2] Unfortunately, there are 49 benchmarks where MUS replication found no u-seed at all. Let us note that 40 of the 49 benchmarks are from the *same family* of benchmarks, called “fdmus”. The MUS benchmarks from the SAT competition consist of several families and benchmarks in a family often have very similar structure. Most of the families contain only few benchmarks, however, there are several larger families and the “fdmus” family is by far the largest one.

Number of Indentified MUSes. We now examine the number of identified MUSes by the evaluated algorithms on individual benchmarks within the time limit of 3600 s. In case of 28 benchmarks, all the algorithms completed the enumeration, and thus found the same number of MUSes. Therefore, we focus here only on the remaining 263 benchmarks.

Scatter plots in Fig. 4 pair-wise compare UNIMUS with its competitors. Each point in the plot shows a result from a single benchmark. The x-coordinate of a point is given be the algorithm that labels the x-axis and the y-coordinate by the algorithm that labels the y-axis. The plots are in a log-scale and hence cannot show points with a zero coordinate, i.e., benchmarks where at least one

[2] Thus, in those benchmarks, SAT solver calls are performed almost only by the shrinking procedure (which uses glucose in our implementation).

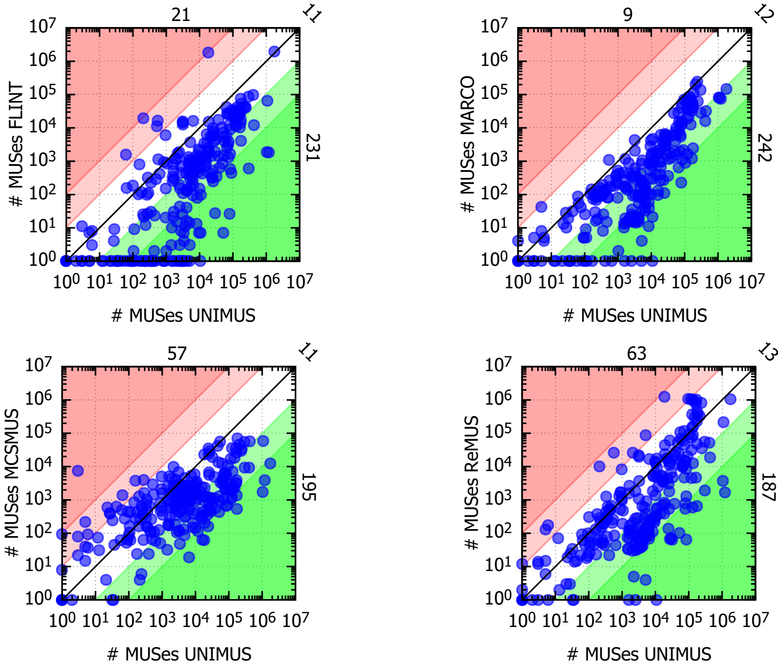

Fig. 4. Scatter plots comparing the number of produced MUSes.

algorithm found no MUS. Therefore, we lifted the points with a zero coordinate to the first coordinate. Moreover, we provide three numbers right/above/in the right corner of the plot, that show the number of points below/above/on the diagonal. For example, UNIMUS found more/less/equal number of MUSes than MARCO in case of 242/9/12 benchmarks. We also use green and red colors to highlight individual orders of magnitude (of 10).

In Fig. 3, we examine the overall *ranking* of the algorithms. In particular, assume that for a benchmark B both UNIMUS and ReMUS found 100 MUSes, MCSMUS found 80 MUSes, and MARCO and FLINT found 50 MUSes. In such a case, UNIMUS and ReMUS share the 1st (best) rank for B, MCSMUS is 3rd, and MARCO and FLINT share the 4th position. For each algorithm, we computed an arithmetic mean of the ranking on all benchmarks. To eliminate the effect of outliers (benchmarks with an extreme ranking), we computed the 5% truncated arithmetic mean, i.e., for each algorithm we discarded the 5% of benchmarks where the algorithm achieved the best and the worst ranking. Moreover, to capture the performance stability of the algorithms in time, we computed the mean for each subsequent 60 s of the computation.

UNIMUS conclusively dominates all its competitors. It maintained the best ranking during the whole time period and gradually improved the ranking towards the final value 1.3. The closest, yet still very distant, competitors are ReMUS and MCSMUS who maintained ranking around 2.75. FLINT and MARCO

achieved the final raking around 3.7. UNIMUS also dominated in the pair-wise comparison. It found more MUSes than all its competitors on an overwhelming majority of benchmarks and, remarkably, the difference was often several orders of magnitude.

References

1. Andraus, Z.S., Liffiton, M.H., Sakallah, K.A.: Cegar-based formal hardware verification: a case study. Technical report, University of Michigan, CSE-TR-531-07 (2007)
2. Arif, M.F., Mencía, C., Ignatiev, A., Manthey, N., Peñaloza, R., Marques-Silva, J.: BEACON: an efficient sat-based tool for debugging $\mathcal{EL}^+$ ontologies. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 521–530. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_32
3. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: IJCAI, pp. 399–404 (2009)
4. Bacchus, F., Katsirelos, G.: Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 70–86. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_5
5. Bacchus, F., Katsirelos, G.: Finding a collection of MUSes incrementally. In: Quimper, C.-G. (ed.) CPAIOR 2016. LNCS, vol. 9676, pp. 35–44. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33954-2_3
6. Bailey, J., Stuckey, P.J.: Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2005. LNCS, vol. 3350, pp. 174–186. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30557-6_14
7. Belov, A., Marques-Silva, J.: Accelerating MUS extraction with recursive model rotation. In: FMCAD, pp. 37–40. FMCAD Inc. (2011)
8. Belov, A., Marques-Silva, J.: MUSer2: an efficient MUS extractor. JSAT **8**, 123–128 (2012)
9. Bendík, J., Beneš, N., Černá, I., Barnat, J.: Tunable online MUS/MSS enumeration. In: FSTTCS, LIPIcs, vol. 65, pages 50:1–50:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
10. Jaroslav Bendík and Ivana Černá. Evaluation of domain agnostic approaches for enumeration of minimal unsatisfiable subsets. In: LPAR, EPiC Series in Computing, vol. 57, pp. 131–142. EasyChair (2018)
11. Bendík, J., Černá, I.: MUST: minimal unsatisfiable subsets enumeration tool. TACAS 2020. LNCS, vol. 12078, pp. 135–152. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_8
12. Bendík, J., Černá, I., Beneš, N.: Recursive online enumeration of all minimal unsatisfiable subsets. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 143–159. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_9
13. Bendík, J., Ghassabani, E., Whalen, M., Černá, I.: Online enumeration of all minimal inductive validity cores. In: Johnsen, E.B., Schaefer, I. (eds.) SEFM 2018. LNCS, vol. 10886, pp. 189–204. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92970-5_12
14. Bendík, J., Meel, K.S.: Approximate counting of minimal unsatisfiable subsets. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 439–462. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_21

15. Biere, A.: Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2018. In: Proceedings of SAT Competition, pp. 13–14 (2018)
16. Bollobás, B., Borgs, C., Chayes, J.T., Kim, J.H., Wilson, D.B.: The scaling window of the 2-sat transition. Random Struct. Algorithms **18**(3), 201–256 (2001)
17. Chen, H., Marques-Silva, J.: Improvements to satisfiability-based Boolean function bi-decomposition. In: VLSI-SoC, pp. 142–147. IEEE (2011)
18. Cohen, O., Gordon, M., Lifshits, M., Nadel, A., Ryvchin, V.: Designers work less with quality formal equivalence checking. In: Design and Verification Conference (DVCon). Citeseer (2010)
19. de la Banda, M.J.G., Stuckey, P.J., Wazny, J.: Finding all minimal unsatisfiable subsets. In: PPDP, pp. 32–43. ACM (2003)
20. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
21. Han, B., Lee, S.-J.: Deriving minimal conflict sets by CS-trees with mark set in diagnosis from first principles. IEEE Trans. Syst. Man Cybern. Part B **29**(2), 281–286 (1999)
22. Hou, A.: A theory of measurement in diagnosis from first principles. AI **65**(2), 281–328 (1994)
23. Hunter, A., Konieczny, S.: Measuring inconsistency through minimal inconsistent sets. In: KR, pp. 358–366. AAAI Press (2008)
24. Ivrii, A., Malik, S., Meel, K.S., Vardi, M.Y.: On computing minimal independent support and its applications to sampling and counting. Constraints **21**(1), 41–58 (2015). https://doi.org/10.1007/s10601-015-9204-z
25. Jannach, D., Schmitz, T.: Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach. Autom. Softw. Eng. **23**(1), 105–144 (2014). https://doi.org/10.1007/s10515-014-0141-7
26. Kilby, P., Slaney, J.K., Thiébaux, S., Walsh, T.: Backbones and backdoors in satisfiability. In: AAAI, pp. 1368–1373. AAAI Press/The MIT Press (2005)
27. Liffiton, M.H., Malik, A.: Enumerating infeasibility: finding multiple MUSes Quickly. In: Gomes, C., Sellmann, M. (eds.) CPAIOR 2013. LNCS, vol. 7874, pp. 160–175. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38171-3_11
28. Liffiton, M.H., Previti, A., Malik, A., Marques-Silva, J.: Fast, flexible MUS enumeration. Constraints **21**(2), 223–250 (2015). https://doi.org/10.1007/s10601-015-9183-0
29. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. JAR **40**(1), 1–33 (2008). https://doi.org/10.1007/s10817-007-9084-z
30. Luo, J., Liu, S.: Accelerating MUS enumeration by inconsistency graph partitioning. Sci. China Inf. Sci. **62**(11), 212104 (2019). https://doi.org/10.1007/s11432-019-9881-0
31. Mencía, C., Kullmann, O., Ignatiev, A., Marques-Silva, J.: On computing the union of MUSes. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 211–221. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_15
32. Kedian, M.: Formulas free from inconsistency: an atom-centric characterization in priest's minimally inconsistent LP. J. Artif. Intell. Res. **66**, 279–296 (2019)
33. Narodytska, N., Bjørner, N., Marinescu, M.-C., Sagiv, M.: Core-guided minimal correction set and core enumeration. In: IJCAI, pp. 1353–1361 (2018). https://www.ijcai.org/

34. Previti, A., Marques-Silva, J.: Partial MUS enumeration. In: AAAI. AAAI Press (2013)
35. Sperner, E.: Ein satz über untermengen einer endlichen menge. Math. Z. **27**(1), 544–548 (1928). https://doi.org/10.1007/BF01171114
36. Stern, R.T., Kalech, M., Feldman, A., Provan, G.M.: Exploring the duality in conflict-directed model-based diagnosis. In: AAAI. AAAI Press (2012)
37. Stuckey, P.J., Sulzmann, M., Wazny, J.: Interactive type debugging in haskell. In: Haskell, pp. 72–83. ACM (2003)
38. Wieringa, S.: Understanding, improving and parallelizing MUS finding using model rotation. In: Milano, M. (ed.) CP 2012. LNCS, pp. 672–687. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33558-7_49