Recursive Online Enumeration of All Minimal Unsatisfiable Subsets

Jaroslav Bendík, Ivana Černá, and Nikola Beneš

Faculty of Informatics, Masaryk University, Brno, Czech Republic
{xbendik,cerna,xbenes3}@fi.muni.cz

Abstract. In various areas of computer science, we deal with a set of constraints to be satisfied. If the constraints cannot be satisfied simultaneously, it is desirable to identify the core problems among them. Such cores are called minimal unsatisfiable subsets (MUSes). The more MUSes are identified, the more information about the conflicts among the constraints is obtained. However, a full enumeration of all MUSes is in general intractable due to the large number (even exponential) of possible conflicts. Moreover, to identify MUSes, algorithms have to test sets of constraints for their simultaneous satisfiability. The type of the test depends on the application domains. The complexity of the tests can be extremely high especially for domains like temporal logics, model checking, or SMT.
In this paper, we propose a recursive algorithm that identifies MUSes in an *online* manner (i.e., one by one) and can be terminated at any time. The key feature of our algorithm is that it minimises the number of satisfiability tests and thus speeds up the computation. The algorithm is applicable to an arbitrary constraint domain and proves to be efficient especially in domains with expensive satisfiability checks. We benchmark our algorithm against the state-of-the-art algorithm Marco on the Boolean and SMT constraint domains and demonstrate that our algorithm really requires less satisfiability tests and consequently finds more MUSes in the given time limits.

1 Introduction

In many different applications we are given a set of constraints (requirements) with the goal to decide whether the set of constraints is satisfiable, i.e., whether all the constraints can hold simultaneously. In case the given set is shown to be unsatisfiable, we might be interested in an analysis of the unsatisfiability. Identification of minimal unsatisfiable subsets (MUSes) is a kind of such analysis. A minimal unsatisfiable subset is a set of constraints that are not simultaneously satisfiable, yet the elimination of any of them makes the set satisfiable. We illustrate the problem on two different applications.

In the *requirements analysis*, the constraints represent requirements on a system that is being developed. Checking for satisfiability (also called *consistency*) means checking whether all the requirements can be implemented at once. If the set of requirements is unsatisfiable, the extraction of MUSes helps to identify and fix the conflicts among the requirements [5, 10].

In some model checking techniques, such as the *counterexample-guided abstraction refinement* (CEGAR) [15], we are dealing with the following question: is the counterexample that was found in an abstract model feasible also in the concrete model? To answer this question, a formula $cex \wedge conc$ encoding both the counterexample *cex* and the concrete model *conc* is built and tested for satisfiability. If the formula is unsatisfiable, then the counterexample is *spurious* and the negation of the formula $cex \wedge conc$ is used to refine the abstract model. Since both *cex* and *conc* are often formed as a conjunction of smaller subformulas, the whole formula can be seen as a set of conjuncts (constraints). Andraus et al. [1, 15] found out that instead of using the negation of $cex \wedge conc$ for the refinement, it is better to identify the MUSes of $cex \wedge conc$ and use the negations of the MUSes to refine the abstract model.

Yet another applications of MUSes arise for example during formal equivalence checking [16], proof based abstraction refinement [29], Boolean function bi-decomposition [12], circuit error diagnosis [24], type debugging in Haskell [36], or proof explanation in symbolic model checking [22].

The individual applications differ in the constraint domain. Perhaps the most widely used are Boolean and SMT constraints; these types of constraints arise for example in the CEGAR workflow. In the requirements analysis, the most common constraints are those expressed in a temporal logic such as *Linear Temporal Logic* [33] or *Computational Tree Logic* [14]. The list of constraint domains in which MUS enumeration finds an application is quite long and new applications still arise. Therefore, we focus on MUS enumeration algorithms applicable in arbitrary constraint domains.

Main contribution All algorithms solving the MUS enumeration problem have to get over two barriers. First, the number of all MUSes can be exponential w.r.t. the number of constraints. Therefore, the complete enumeration of all MUSes can be intractable. To overcome this limitation we present an algorithm for *online* MUS enumeration, i.e., an algorithm that enumerates MUSes one by one and can be terminated at any time.

Second, algorithms for MUS enumeration need to test whether a given set of constraints is satisfiable. This is typically a very hard task and it is thus desirable to minimise the overall number of satisfiability queries. To reduce the number of performed satisfiability queries, the majority of the state-of-the-art algorithms (for their detailed description see Section 4) try to exploit specific properties of particular constraint domains. Most of the algorithms were evaluated only in the SAT domain (the domain of Boolean constraints). The SAT domain enjoys properties that can be used to significantly reduce the number of satisfiability queries and the state-of-the-art algorithms are thus very efficient in this domain. However, this might not be the case in domains for which no such domain specific properties exist. Here, we present a novel algorithm that exploits both the domain specific as well as domains agnostics properties of the MUS enumeration problem. First, the algorithm employs, as a black-box subroutine, a domain specific single MUS extraction algorithm which allows it to exploit specific properties of particular domains. Second, it recursively searches for MUSes in

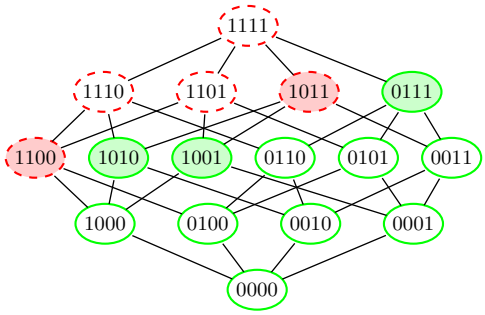


Fig. 1: Illustration of the power set of the set C of constraints from the Example 1. We encode individual subsets of C as bitvectors, e.g. the subset $\{c_1, c_3, c_4\}$ is written as 1011. The subsets with dashed border are the unsatisfiable subsets and the others are satisfiable subsets. The MUSes and MSSes are filled with a background colour.

smaller and smaller subsets of the given set of constraints which allows it to directly reduce the number of performed satisfiability queries.

2 Preliminaries and Problem Statement

We are given a finite set of constraints C with the property that each subset of C is either *satisfiable* or *unsatisfiable*. The notion of satisfiability may vary in different constraint domains. The only assumption is that if a set X, $X \subseteq C$, is unsatisfiable, then all supersets of X are unsatisfiable as well. The sets of interests are defined as follows.

Definition 1 (MUS, MSS, MCS). *Let C be a finite set of constraints and let $N \subseteq C$. N is a* minimal unsatisfiable subset *(MUS) of C if N is unsatisfiable and $\forall c \in N : N \setminus \{c\}$ is satisfiable. N is a* maximal satisfiable subset *(MSS) of C if N is satisfiable and $\forall c \in C \setminus N : N \cup \{c\}$ is unsatisfiable. N is a* minimal correction set *(MCS) of C if $C \setminus N$ is a MSS of C.*

The maximality concept used in the definition of a MSS is the set maximality and not the maximum cardinality as in the MaxSAT problem. Thus a constraint set C can have multiple MSSes with different cardinalities.

Example 1. Assume that we are given a set $C = \{c_1, c_2, c_3, c_4\}$ of four Boolean satisfiability constraints $c_1 = a$, $c_2 = \neg a$, $c_3 = b$, and $c_4 = \neg a \vee \neg b$. Clearly, the whole set is unsatisfiable as the first two constraints are negations of each other. There are two MUSes of C, namely $\{c_1, c_2\}$, $\{c_1, c_3, c_4\}$. There are three MSSes of C, namely $\{c_1, c_4\}$, $\{c_1, c_3\}$, and $\{c_2, c_3, c_4\}$. Finally, there are three MCSes of C, namely $\{c_2, c_3\}$, $\{c_2, c_4\}$, and $\{c_1\}$. This example is illustrated in Fig. 1.

Another concept used in our work are the so-called *critical constraints* that are defined as follows:

Definition 2 (critical constraint). *Let C be a finite set of constraints and let $N \subseteq C$ be its unsatisfiable subset. A constraint $c \in N$ is a* critical constraint *for N if $N \setminus \{c\}$ is satisfiable.*

Note that N is a MUS of C if and only if each $c \in N$ is critical for N. Furthermore, if c is a critical constraint for C then c has to be contained in every unsatisfiable subset of C, especially in every MUS of C. Also, note that if S is a MSS of C and $\overline{S} = C \setminus S$ its complement (i.e. a MCS of C), then each $c \in \overline{S}$ is critical for $S \cup \{c\}$.

Example 2. We illustrate the concept of critical constraints on two sets, N and C, where C is the same set as in Example 1, and $N = C \setminus \{c_2\}$. The constraint c_1 is the only critical constraint for C whereas N has three critical constraints: c_1, c_3, and c_4.

Problem Formulation Given a set of constraints C, enumerate all minimal unsatisfiable subsets of C in an online manner while minimising the number of constraints satisfiability queries. Moreover, we require an approach that is applicable to an arbitrary constraint domain.

3 Algorithm

We start with some observations about the MUS enumeration problem and describe the main concepts used in our algorithm.

The algorithm is given an unsatisfiable set of constraints C. To find all MUSes, the algorithm iteratively determines satisfiability of subsets of C. Initially, only the satisfiability of C is determined and at the end, satisfiability of all subsets of C is determined. The algorithm maintains a set *Unexplored* containing all subsets of C whose satisfiability is not determined yet. Recall that if a set of constraints is satisfiable then all its subsets are satisfiable as well. Therefore, if the algorithm determines some $N \subseteq C$ to be satisfiable, then not only N but also all of its subsets, denoted by $sub(N)$, become explored (i.e. are removed from the set *Unexplored*). Dually, if N is unsatisfiable then all of its supersets, denoted by $sup(N)$, are unsatisfiable and become explored.

Since there are exponentially many subsets of C, it is intractable to represent the set *Unexplored* explicitly. Instead, we use a symbolic representation that is common in contemporary MUS enumeration algorithms [27, 11, 22]. We encode $C = \{c_1, c_2, \ldots, c_n\}$ by using a set of Boolean variables $X = \{x_1, x_2, \ldots, x_n\}$. Each valuation of X then corresponds to a subset of C. This allows us to represent the set of unexplored subsets *Unexplored* using a Boolean formula $f_{Unexplored}$ such that each model of $f_{Unexplored}$ corresponds to an element of *Unexplored*. The formula is maintained as follows:

- Initially $f_{Unexplored} = \mathit{True}$ since all of $\mathcal{P}(C)$ are unexplored.
- To remove a satisfiable set $N \subseteq C$ and all its subsets from the set *Unexplored* we add to $f_{Unexplored}$ the clause $\bigvee_{i:c_i \notin N} x_i$.

- Symmetrically, to remove an unsatisfiable set $N \subseteq C$ and all its supersets from the set *Unexplored* we add to $f_{Unexplored}$ the clause $\bigvee_{i:c_i \in N} \neg x_i$.

To get an element of *Unexplored*, we ask a SAT solver for a model of $f_{Unexplored}$.

Example 3. Let us illustrate the symbolic representation on $C = \{c_1, c_2, c_3\}$. If all subsets of C are unexplored then $f_{Unexplored} = True$. If $\{c_1, c_3\}$ is determined to be unsatisfiable and $\{c_1, c_2\}$ to be satisfiable, then $f_{Unexplored}$ is updated to $True \wedge (\neg x_1 \vee \neg x_3) \wedge (x_3)$.

One of the approaches (see e.g. [26, 11, 34]) how to find a MUS of C is to find an unexplored unsatisfiable subset, called a *seed*, and then use any algorithm that finds a MUS of the seed (this algorithm is often denoted as a *shrink* procedure). In our algorithm we use a black-box subroutine for shrinking. This allows us to always employ the best available, domain specific, single MUS extraction algorithm.

The key question is how to find an unexplored unsatisfiable subset (a seed). Due to the monotonicity of the satisfiability of individual subsets (w.r.t. subset inclusion), satisfiable subsets are typically smaller and, dually, unsatisfiable subsets are more concentrated among the larger subsets. Therefore, we search for seeds among *maximal unexplored subsets*. A set S_{max} is a maximal unexplored subset of C iff $S_{max} \subseteq C$, $S_{max} \in Unexplored$, and each of the proper supersets of S_{max} is explored. The maximal unexplored subsets correspond to the *maximal models* of $f_{Unexplored}$. Thus, in order to get a maximal unexplored subset S_{max}, we ask a SAT solver for such a model. If S_{max} is unsatisfiable, we use it as a seed for the shrinking procedure and compute a MUS of C.

The idea of searching for seeds among the maximal unexplored subsets is already used in some contemporary MUS enumeration algorithms [27, 34, 11, 22]. However, none of the algorithms specify which maximal unexplored subset should be used for finding a seed. They just ask a SAT solver for an arbitrary maximal model of $f_{Unexplored}$ (maximal unexplored subset). We found that the choice of maximal unexplored subset to be used is very important. The complexity of the shrinking procedure, in general, depends on the cardinality (the number of constraints) of the seed. Thus, an ideal option would be to search for a seed among the maximal unexplored subsets with the minimum cardinality, i.e. to find a *minimum maximal model* of $f_{Unexplored}$. However, finding such a model is very expensive, especially compared to finding an arbitrary maximal model of $f_{Unexplored}$. In order to find an arbitrary maximal model, we can just instruct the SAT solver to use *True* as a default polarity of variables during solving (this can be done e.g. in the miniSAT [20] solver). On the other hand, finding a minimum maximal model of $f_{Unexplored}$ is a hard optimisation problem.

We propose a way of finding seeds that are relatively small, yet cheap to be found. To find the first seed we are repeatedly asking the SAT solver for an arbitrary maximal unexplored subset of C until we obtain some unsatisfiable maximal unexplored subset S_{max}. Then, we use S_{max} as the first seed and shrink it to the first MUS S_{mus}. In order to find the next seed, we use a more sophisticated approach. Instead of searching for a seed among the maximal unexplored subsets

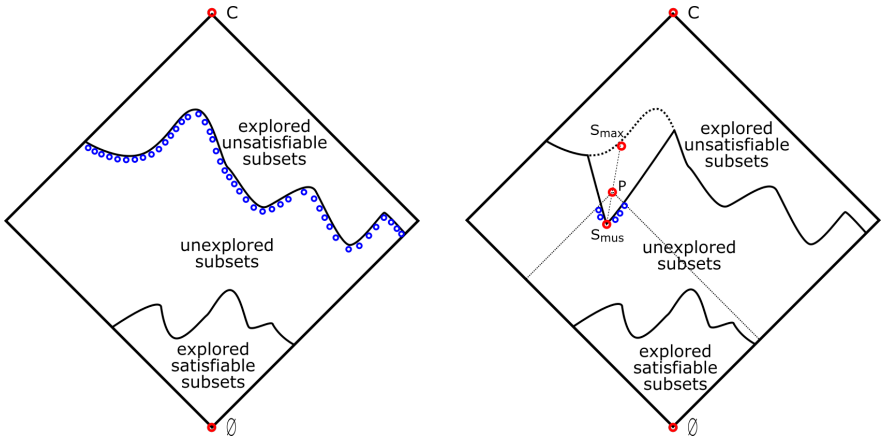


Fig. 2: Illustration of our seed selection approach. Figure 2a illustrates the division of subsets of C into explored satisfiable, explored unsatisfiable, and unexplored subsets. The blue circle nodes represent the maximal unexplored subsets of C. Figure 2b shows a previously used seed S_{max}, a MUS S_{mus} that was found based on the seed, a set P such that $S_{mus} \subseteq P \subseteq S_{max}$, and maximal unexplored subsets of P (blue circle nodes).

of C, we restrict the search space so that the next seed is smaller than the previous one. In particular, we choose a set P such that $S_{mus} \subseteq P \subseteq S_{max}$ and search for the new seed among the maximal unexplored subsets of P. Note that the maximal unexplored subsets of P do not have to be maximal unexplored subsets of C. Furthermore, P is necessarily unsatisfiable and all seeds found within it are necessarily smaller than the previous seed S_{max} since $P \subseteq S_{max}$. By choosing the next seed among the maximal unexplored subsets of P, we de facto solve the problem in a recursive manner. Instead of finding a new MUS of C, we find a MUS of P, which is necessarily also a MUS of C. Each next seed is found based on the previous one, i.e. we keep to recursively reducing the search space as far as we can. Once we end up in a subset P of C such that the whole $\mathcal{P}(P)$ is explored, we backtrack from the recursion. The approach is illustrated in Fig. 2.

In our algorithm we also use critical constraints. For mining critical constraints we use the maximal unexplored subsets that are satisfiable. Every satisfiable maximal unexplored subset S_{max} of C is a maximal satisfiable subset (MSS) of C as every superset of S_{max} is explored. If it were satisfiable then due to monotonicity S_{max} should also be explored (which it is not). Thus, for every $c \in C \setminus S_{max}$ it holds that $S_{max} \cup \{c\}$ is unsatisfiable and c is a critical constraint for $S_{max} \cup \{c\}$.

The critical constraints are used in two different situations. The first situation arises when we are searching for a seed and the selected maximal unexplored subset S_{max} of C is satisfiable. In such a case we can try to pick another maximal

unexplored subset of C and check it for satisfiability. However, for reasons mentioned above, we try to search for small seeds. Therefore we recursively search for a maximal unexplored subset of $S_{max} \cup \{c\}$, where c is a critical constraint for $S_{max} \cup \{c\}$. The set $S_{max} \cup \{c\}$ is definitely unsatisfiable and its cardinality is not greater than the cardinality of C.

Many modern shrinking algorithms [2, 8, 31] use critical constraints to speed up their computation. Every MUS of C has to contain all the critical constraints for C and this helps the shrinking procedure to narrow the search space. The critical constraints for C have to be delivered to the shrinking algorithm together with the seed. Our algorithm can compute and accumulate critical constraints effectively while recursively traversing the space. If X and Y are two sets of constraints such that $X \supseteq Y$, then every critical constraint for X is also a critical constraint for Y. The algorithm thus can utilise the known critical constraints even in its recursive part.

3.1 Description of the Algorithm

The pseudocode of our algorithm is shown in Algorithm 1. The computation of the algorithm starts with an initialisation phase followed by a call of the procedure `FindMUSes`, which is the core procedure of our algorithm.

The procedure `FindMUSes` has two input parameters: S and *criticals*. S is an unsatisfiable set of constraints and the procedure outputs MUSes of S. The set *criticals* contains (currently known) critical constraints for S and is used for the shrinking procedure. In each iteration, the procedure `FindMUSes` picks a maximal unexplored subset S_{max} of S and tests it for satisfiability. If S_{max} is satisfiable, then it is guaranteed to be a MSS of S. Thus, the complement $S_{mcs} = S \setminus S_{max}$ of S_{max} is an MCS of S and it can be used to obtain critical constraints. If $|S_{mcs}| = 1$, then the single constraint that forms S_{mcs} is guaranteed to be a critical constraint for S and it is thus added into *criticals*. Otherwise, the procedure recursively calls itself on $(S_{max} \cup \{c\}, \mathit{criticals} \cup \{c\})$ for each $c \in S_{mcs}$ since each such c is guaranteed to be a critical constraint for $S_{max} \cup \{c\}$.

In the other case, when S_{max} is unsatisfiable, then S_{max} is shrunk to a MUS S_{mus} (note that the set *criticals* of critical constraints is provided to the shrinking procedure). The newly computed S_{mus} is used to reduce the dimension of the space in which another MUSes are searched for. Namely, the procedure picks some P, $S_{mus} \subset P \subset S_{max}$, and recursively calls itself on P. After the recursive call terminates, the procedure continues with the next iteration.

The main idea behind the recursion is to search for MUSes of sets smaller than S and thus lower the complexity of performed operations. Naturally, there is a trade-off between the complexity of operations and the expected number of MUSes occurring in the chosen subspace and thus it might be very tricky to find an optimal P. In our algorithm we choose P so that $|P| = 0.9 \cdot |S_{max}|$, where $|P|$ and $|S_{max}|$ are cardinalities of the two sets, respectively. We form P by adding a corresponding number of constraints from S_{max} to S_{mus}. Note that it might happen that $|S_{mus}| \geq 0.9 \cdot |S_{max}|$; in such a case the algorithm skips the recursion call and continues with the next iteration.

Algorithm 1: ReMUS

```
1  Function Init(C):
       input  : an unsatisfiable set of constraints C
2      Unexplored ← P(C)
3      FindMUSes(C, ∅)
1  Function FindMUSes(S, criticals):
2      while Unexplored ∩ P(S) ≠ ∅ do
3          S_max ← a maximal unexplored subset of S
4          if S_max is satisfiable then
5              Unexplored ← Unexplored \ Sub(S_max)
6              S_mcs ← S \ S_max
7              if |S_mcs| = 1 then
8                  criticals ← criticals ∪ S_mcs
9              else
10                 for each c ∈ S_mcs do
11                     FindMUSes(S_max ∪ {c}, criticals ∪ {c})
12         else
13             S_mus ←Shrink(S_max, criticals)
14             output S_mus
15             Unexplored ← Unexplored \ (Sup(S_mus) ∪ Sub(S_mus))
16             if |S_mus| < 0.9 · |S_max| then
17                 P ← subset such that S_mus ⊂ P ⊂ S_max, |P| = 0.9 · |S_max|
18                 FindMUSes (P, criticals)
```

The set *Unexplored* is updated appropriately during the whole computation. Note that the set *Unexplored* is shared among the individual recursive calls; in particular if the algorithm determines some subset S to be unsatisfiable then all of its supersets (w.r.t. the original search space) are deduced to be unsatisfiable. On the other hand, the maximal unexplored subsets (and their complements) are local and are defined with respect to the current search space.

Correctness The algorithm outputs only the results of *shrinking* which is assumed to be a correct MUS extraction procedure. Each MUS is produced only once since only unexplored subsets are shrunk and each MUS is removed from the set *Unexplored* immediately after producing. Only subsets whose status is known are removed from the set *Unexplored* thus no MUS is excluded from the computation. The algorithm terminates and all MUSes are found since the size of *Unexplored* is reduced after every iteration.

4 Related Work

The list of existing approaches to the MUS enumeration problem is short, especially compared to the amount of work dealing with a single MUS extraction [6–9, 30, 32]. Moreover, existing algorithms for the MUS enumeration are tailored

mainly to Boolean constraints [21, 23, 3, 2] and cannot be applied to other constraints. The approaches that focus on MUS enumeration in general constraint systems can be divided into two categories: approaches that compute MUSes directly and those that rely on the *hitting set duality*.

Direct MUS enumeration
The early algorithms were based on explicit enumeration of every subset of the unsatisfiable constraint system. As far as we know, the MUS enumeration was pioneered by Hou [25] in the field of diagnosis. Hou's algorithm checks every subset for satisfiability starting with the whole set of constraints and exploring its power set in a tree-like structure. Also, some pruning rules that allow skipping irrelevant branches are presented. This approach was revisited and further improved by Han and Lee [24] and by de la Banda et al. [17]. Another approach using step-by-step powerset exploration was recently proposed by Bauch et al. [5]. The authors of this work focus on constraints expressed using LTL formulas; however, their algorithm can be used for any type of constraints. Explicit exploration of the power set is the bottleneck of all of the above mentioned algorithms as the size of the power set is exponential to the number of constraints in the system.

Liffiton et al. [26] and Silva et al. [34] developed independently two nearly identical algorithms: MARCO [26] and eMUS [34]. Both algorithms were later merged and presented [27] under the name of MARCO. Among the existing MUS enumeration algorithms, MARCO is perhaps the one most similar to ReMUS. It uses symbolic representation of the power set and is able to produce MUSes incrementally during its computation in a relatively steady rate. In order to find individual MUSes, it iteratively picks maximal unexplored subsets of the original set of constraints and checks them for satisfiability. The unsatisfiable subsets are shrunk, using a black-box procedure, into MUSes. Contrary to ReMUS, MARCO does not tend to reduce the size of the sets to be shrunk and thus to directly reduce the number of performed satisfiability checks. Instead, it assumes that the black-box shrinking procedure would do the trick. MARCO is very efficient in constraint domains for which efficient shrink procedures exist. However, in the other domains, it is less efficient. This is mainly due to the fact that it shrinks the maximal unexplored subsets of the original set of constraints, i.e. it shrinks relatively large sets.

In our previous work [11], we have presented the algorithm TOME that also produces MUSes in an online manner. It iteratively uses binary search to find the so-called local MUSes/MSSes. Each local MUS/MSS is optionally, based on its size (cardinality), shrunk/grown to a global MUS/MSS. TOME tries to predict the complexity of performing the shrinking/growing procedure and only those shrinks/grows that seem to be easy to perform are actually performed. TOME is very efficient in constraint domains for which no efficient shrinking and growing procedure exist. On the other hand, in domains like Boolean constraints, the effort needed to find local MUSes and MSSes outweighs the effort needed to perform the shrinks and grows.

Hitting set duality based approaches
There is a well known relationship between MUSes and MCSes based on the

concept of *hitting sets*. Given a collection Ω of sets, a hitting set H for Ω is a set such that $\forall S \in \Omega : H \cap S \neq \emptyset$. A hitting set is called *minimal* if none of its proper subsets is a hitting set. If C is a set of constraints and $N \subseteq C$, then the *minimal hitting set duality* [35] claims that N is a MUS of C iff N is a minimal hitting set of the set of all the MCSes of C.

The hitting set duality is used for example in CAMUS [28] and DAA [4]. CAMUS works in two phases. It first computes all MCSes of a given constraint set and then finds all MUSes by computing all minimal hitting sets of these MCSes. A significant shortcoming of CAMUS is that the first phase can be intractable as the number of MCSes can be exponential in the number of constraints and all MCSes must be found before the first MUS can be produced.

The algorithm DAA [4] is able to produce some MUSes before the enumeration of MCSes is completed. DAA starts each iteration with computing a minimal hitting set H of currently known MCSes and tests H for satisfiability. If H is unsatisfiable, it is guaranteed to be a MUS. In the other case, H is *grown* into a MSS whose complement is a MCS, i.e. the set of known MCSes is enlarged. As in the case of MARCO, DAA can use any existing algorithm for a single MSS/MCS extraction to perform the grow.

MARCO, CAMUS and DAA were experimentally compared in the Boolean constraints domain [27] and CAMUS has shown to be the fastest in enumerating all MUSes in the tractable cases. However, in the intractable cases, MARCO was able to produce at least some MUSes, while CAMUS often got stuck in the phase of MCSes enumeration. DAA was much slower than CAMUS in the case of complete MUSes enumeration and also slower than MARCO in the case of partial MUS enumeration. The main drawbacks of DAA are the complexity of computing minimal hitting sets and no guarantee on the rate of MUS production.

Bacchus and Katsirelos proposed a MUS enumeration algorithm called MCS-MUS-BT [3] which is also based on recursion and uses MCSes to extract critical constraints. However, the algorithm is tailored for the SAT domain and, thus, cannot be applied in an arbitrary constraint domain. Moreover, the computation of MCSes is an integral part of MCS-MUS-BT, and the MCSes are computed in a different way taking up to linearly many satisfiability checks to compute each MCS. MCS-MUS-BT does not use black-box shrinking procedures and the recursion is not driven by previously found MUSes.

5 Implementation

We implemented ReMUS into a publicly available tool[1]. The tool currently supports three different constraint domains: SAT (Boolean constraints), SMT, and LTL. It employs several external tools. In particular, it uses the SAT solver miniSAT [20] for maintaining $f_{Unexplored}$, and miniSAT is also used as a satisfiability solver in the SAT domain. The tools Z3 [18] and SPOT [19] are used as satisfiability solvers in the SMT and LTL domains, respectively. Moreover, our

[1] https://www.fi.muni.cz/ xbendik/remus/

tool uses the single MUS extractor MUSer2 [8] as a black-box shrink subroutine in the SAT domain. In the other domains, we use our custom implementation of the shrinking procedures.

6 Experimental Evaluation

Here, we report results of our experimental evaluation. Besides evaluating ReMUS, we also provide a comparison with the latest tool implementation[2] of the state-of-the-art MUS enumeration algorithm MARCO [27]. The comparison is done in the SAT and SMT domains since these are the domains supported by the MARCO tool. Note that MARCO uses the same external procedures as ReMUS, i.e. a satisfiability solver, a shrinking procedure, and a SAT solver for maintaining unexplored subsets. All these external procedures are implemented in the MARCO tool in the same way as in the ReMUS tool, i.e. using miniSAT [20], Z3 [18], and MUSer2 [8].

There are three main criteria for the comparison: 1) the number of output MUSes within a given time limit, 2) the number of satisfiability checks required to output individual MUSes, and 3) the time required to output individual MUSes.

6.1 Benchmarks and Experimental Setup

The experiments in the SAT domain were conducted on a collection of 292 Boolean CNF benchmarks that were taken from the MUS track of the SAT 2011 competition[3]. The benchmarks range in their size from 70 to 16 million constraints and use from 26 to 4.4 million variables. This collection of benchmarks has been already used in several papers that focus on the problem of MUS enumeration, see e.g. [11, 26–28]. In the SMT domain, we used a set of 433 benchmarks that were used in the work by Griggio et al. [13]. The benchmarks were selected from the library SMT-LIB[4], and include instances from the QF_UF, QF_IDL, QF_RDL, QF_LIA and QF_LRA divisions. The size of the benchmarks ranges from 5 to 145422 constraints.

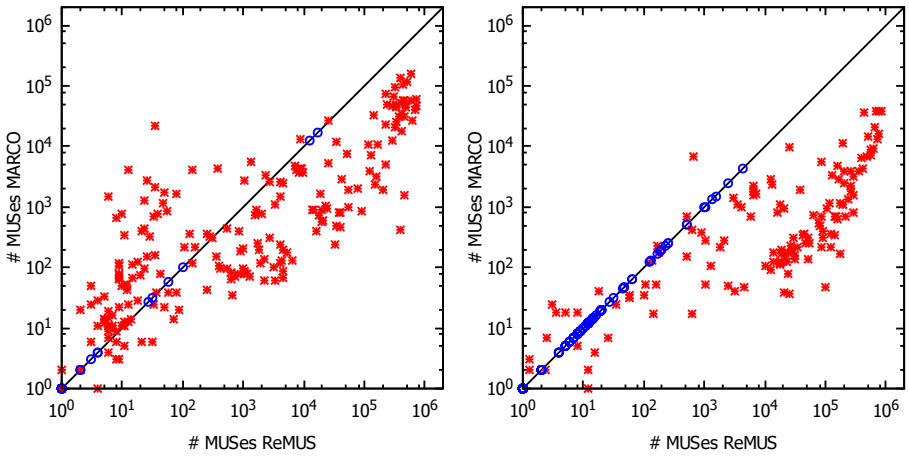
The experiments were run on an Intel(R) Xeon (R) CPU E5-2630 v2, 2.60GHz, 125 GB memory machine running Arch Linux 4.9.40-l-lts. All experiments were run using a time limit of 3600 seconds. Complete results are available at `https://www.fi.muni.cz/~xbendik/remus/`.

7 Experimental Results

7.1 Number of Output MUSes

In this section, we examine the performance of evaluated algorithms in terms of number of produced MUSes within the given time limit of 3600 seconds. Due to

[2] `https://sun.iwu.edu/~mliffito/marco/`
[3] `http://www.cril.univ-artois.fr/SAT11/`
[4] `http://www.smt-lib.org/`



(a) SAT domain. (b) SMT domain.

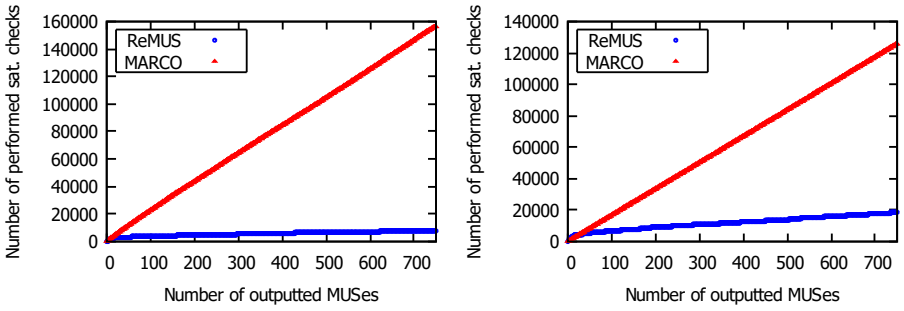
Fig. 3: Scatter plots comparing the number of produced MUSes. Blue points represent the benchmarks where both algorithms finished the computation.

the potentially exponentially many MUSes in each instance, the complete MUS enumeration is generally intractable. Moreover, even producing a single MUS can be intractable for larger instances as it naturally includes solving the satisfiability problem, which is hard to solve in the SAT and SMT domains. Within the given time limit, both algorithms found more than two MUSes in only 216 SAT and 238 SMT instances. Furthermore, both algorithms finished the computation in only 24 SAT and 245 SMT instances.

Figure 3 provides scatter plots that compare both evaluated algorithms on individual benchmarks in the SAT and SMT domains. Each point in the plot represents the result achieved by the two compared algorithms on one particular instance; one algorithm determines the position on the vertical axis and the other one the position on the horizontal axis. MARCO found strictly more MUSes than ReMUS in 76 SAT and 15 SMT instances. On the other hand, ReMUS found strictly more MUSes than MARCO in 162 SAT and 118 SMT instances. Note that in the SMT domain, ReMUS was often better than MARCO by two orders of magnitude.

7.2 Performed Checks per MUS

In this section, we focus on the main optimisation criterion of our algorithm: the number of checks required to output individual MUSes. This number differs for different benchmarks since individual benchmarks vary in many aspects such as the size of the benchmarks and the size of the MUSes contained in the benchmarks. Therefore, we focus on average values. Plots in Fig. 4 show the average number of performed satisfiability checks required to output the first



(a) SAT domain. (b) SMT domain.

Fig. 4: Plots showing the average number of performed satisfiability checks required to output individual MUSes.

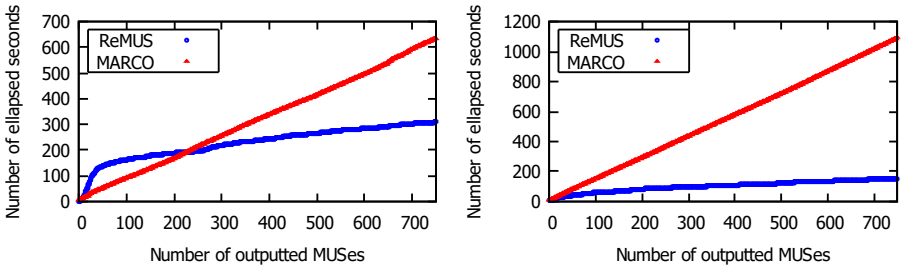
750 MUSes. A point with coordinates (x, y) states that the algorithm needed to perform y satisfiability checks on average in order to output the first x MUSes. We used only a subset of the benchmarks to compute the average values since only for some benchmarks both algorithms found at least 750 MUSes. In particular, 70 and 51 benchmarks were used to compute the average values in the SAT and SMT domains, respectively.

ReMUS is clearly superior to MARCO in the number of satisfiability checks required to output individual MUSes. This happens both due to the fact that ReMUS gradually, in a recursive way, reduces the dimension of the search space (and thus shrink smaller seeds), as well as due the fact that ReMUS mines and accumulates critical constraints to speed up the shrinking procedures.

7.3 Elapsed Time per MUS

The fact that ReMUS requires less satisfiability checks than MARCO to output individual MUSes does not necessarily mean that it is also faster than MARCO in producing individual MUSes. The time spent by ReMUS to maintain the recursive calls while trying to save some satisfiability checks might not be worth it if the checks are easy to perform. We need to answer a domain specific question: is the price of performing satisfiability checks high enough?

To answer this question for the SAT and SMT domains, we took the 70 SAT and 51 SMT benchmarks in which both algorithms produced at least 750 MUSes and computed the average amount of time required to output individual MUSes. The results are shown in Fig. 5. A point with coordinates (x, y) states that in order to output the first x MUSes the algorithm required y seconds on average. In the SMT domain, ReMUS is significantly faster from the very beginning of the computation. In the SAT domain, MARCO is faster during the first three minutes, yet afterwards ReMUS becomes much faster.



(a) SAT domain. (b) SMT domain.

Fig. 5: Plots showing the average amount of time required to output individual MUSes.

7.4 Evaluation

Experimental results demonstrate that ReMUS outperformed MARCO on almost all the SMT instances and on a majority of the SAT instances. However, on some SAT instances, ReMUS was quite struggling, especially at the beginning of the computation. Here, we point out three characteristics of benchmarks/domains that affect the performance of ReMUS.

First, ReMUS tends to minimise the number of performed satisfiability checks. Therefore, the higher the complexity of the satisfiability checks, the more is the tendency to minimise the number of performed checks worth it. Second, the motivation behind finding small seeds for shrinking procedure is based on the fact that, in general, the larger the seed is, the more satisfiability checks are required. However, some constraint domains might enjoy domain specific properties that allow to shrink the seed very efficiently, regardless of the size of the seed. In particular, the CNF form of Boolean (SAT) formulas allows to significantly reduce the number of performed satisfiability checks [8, 32, 6]. Finally, the reduction of the search space is driven by the previously found MUSes. In order to perform deep recursion calls, the input instance has to contain many MUSes. Moreover, there have to be some similar MUSes, i.e. there has to be a subset that is relatively small and yet contains several MUSes.

In the SMT domain, the shrinking procedures are currently not so advanced as in the SAT domain, and the complexity of the satisfiability checks in the SMT domain is often larger than in the SAT domain. Thus, even a small reduction of the size of the seeds leads to a notable improvement in the overall efficiency. On the other hand, in the SAT domain, either a significant reduction of the size of the seeds (i.e. deep recursion calls) or a large number of cumulated critical constraints is required to speed up the shrinking.

8 Conclusion

We have presented the algorithm ReMUS for online enumeration of MUSes that is applicable to an arbitrary constraint domain. We observed that the

time required to output individual MUSes generally correlates with the number of satisfiability checks performed to output the MUSes. The novelty of our algorithm lies in exploiting both the domain specific as well as domain agnostic properties of the MUS enumeration problem to reduce the number of performed satisfiability checks, and thus also reduce the time required to output individual MUSes. The main idea of the algorithm is to recursively search for MUSes in smaller and smaller subsets of a given set of constraints. Moreover, the algorithm cumulates critical constraints and uses them to speed up single MUS extraction subroutines. We have experimentally compared ReMUS with the state-of-the-art MUS enumeration algorithm MARCO in the SAT and SMT domains. The results show that the tendency to minimise the number of performed satisfiability checks leads to a significant improvement over the state-of-the-art.

References

1. Zaher S Andraus, Mark H Liffiton, and Karem A Sakallah. Cegar-based formal hardware verification: A case study. *Ann Arbor*, 2007.
2. Fahiem Bacchus and George Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *CAV (2)*, 2015.
3. Fahiem Bacchus and George Katsirelos. Finding a collection of MUSes incrementally. In *CPAIOR*, 2016.
4. James Bailey and Peter J Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *Practical Aspects of Declarative Languages*. 2005.
5. Jiří Barnat, Petr Bauch, Nikola Beneš, Luboš Brim, Jan Beran, and Tomáš Kratochvíla. Analysing sanity of requirements for avionics systems. *Formal Aspects of Computing*, 2016.
6. Anton Belov, Marijn Heule, and João Marques-Silva. MUS extraction using clausal proofs. In *SAT*, 2014.
7. Anton Belov and João Marques-Silva. Accelerating MUS extraction with recursive model rotation. In *FMCAD*, 2011.
8. Anton Belov and Joao Marques-Silva. MUSer2: An efficient MUS extractor. *Journal on Satisfiability, Boolean Modeling and Computation*, 2012.
9. Anton Belov and João P. Marques Silva. Minimally unsatisfiable boolean circuits. In *SAT*, 2011.
10. Jaroslav Bendík. Consistency checking in requirements analysis. In *ISSTA*, 2017.
11. Jaroslav Bendík, Nikola Benes, Ivana Cerná, and Jiri Barnat. Tunable online MUS/MSS enumeration. In *FSTTCS*, 2016.
12. Huan Chen and João Marques-Silva. Improvements to satisfiability-based boolean function bi-decomposition. In *VLSI-SoC*, 2011.
13. Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Computing small unsatisfiable cores in satisfiability modulo theories. *J. Artif. Intell. Res.*, 2011.
14. Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, 1981.
15. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.
16. Orly Cohen, Moran Gordon, Michael Lifshits, Alexander Nadel, and Vadim Ryvchin. Designers work less with quality formal equivalence checking. In *Design and Verification Conference (DVCon)*, 2010.

17. Maria Garcia de la Banda, Peter J. Stuckey, and Jeremy Wazny. Finding all minimal unsatisfiable subsets. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, 2003.
18. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.
19. Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 — A framework for LTL and ω-automata manipulation. In *ATVA*, 2016.
20. Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, 2003.
21. Rafael M. Gasca, Carmelo Del Valle, María Teresa Gómez López, and Rafael Ceballos. NMUS: structural analysis for improving the derivation of all muses in overconstrained numeric csps. In *CAEPIA*, 2007.
22. Elaheh Ghassabani, Michael W. Whalen, and Andrew Gacek. Efficient generation of all minimal inductive validity cores. In *FMCAD*, 2017.
23. John Gleeson and Jennifer Ryan. Identifying minimally infeasible subsystems of inequalities. *INFORMS Journal on Computing*, 1990.
24. Benjamin Han and Shie-Jue Lee. Deriving minimal conflict sets by cs-trees with mark set in diagnosis from first principles. *IEEE Trans. Systems, Man, and Cybernetics, Part B*, 1999.
25. Aimin Hou. A theory of measurement in diagnosis from first principles. *Artif. Intell.*, 1994.
26. Mark H. Liffiton and Ammar Malik. Enumerating infeasibility: Finding multiple MUSes quickly. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, 2013.
27. Mark H. Liffiton, Alessandro Previti, Ammar Malik, and Joao Marques-Silva. Fast, flexible MUS enumeration. *Constraints*, 2015.
28. Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 2008.
29. Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *TACAS*, 2003.
30. Alexander Nadel. Boosting minimal unsatisfiable core extraction. In *FMCAD*, 2010.
31. Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Efficient MUS extraction with resolution. In *FMCAD*, 2013.
32. Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Accelerated deletion-based extraction of minimal unsatisfiable cores. *JSAT*, 2014.
33. Amir Pnueli. The temporal logic of programs. In *FOCS*, 1977.
34. Alessandro Previti and João Marques-Silva. Partial MUS enumeration. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA.*, 2013.
35. Raymond Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 1987.
36. Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Interactive type debugging in haskell. In *Haskell*, 2003.