On Computing the Union of MUSes *

Carlos Mencía[1], Oliver Kullmann[2], Alexey Ignatiev[3,4], and Joao Marques-Silva[3]

[1] University of Oviedo, Gijón, Spain menciacarlos@uniovi.es
[2] Swansea University, Swansea, UK o.kullmann@swansea.ac.uk
[3] Faculty of Science, University of Lisbon, Portugal
{aignatiev,jpms}@ciencias.ulisboa.pt
[4] ISDCT SB RAS, Irkutsk, Russia

Abstract. This paper considers unsatisfiable CNF formulas and addresses the problem of computing the union of the clauses included in some minimally unsatisfiable subformula (MUS). The union of MUSes represents a useful notion in infeasibility analysis since it summarizes all the causes for the unsatisfiability of a given formula. The paper proposes a novel algorithm for this problem, developing a refined recursive enumeration of MUSes based on powerful pruning techniques. Experimental results indicate the practical suitability of the approach.

1 Introduction

A growing number of practical applications of Boolean Satisfiability (SAT) solvers is related to the analysis of overconstrained formulas, with different instantiations of model-based diagnosis (MBD) representing one of the most visible classes of applications. Examples of diagnosis problems include software fault localization [11,12], spreadsheet debugging [8,9], design debugging [28,31], type error debugging [32], and axiom pinpointing [29], among others. In most settings, the core reasoning services used in the analysis of overconstrained formulas include extraction and enumeration of minimally unsatisfiable subformulas (MUSes), and also minimal correction subsets (MCSes). Tightly related with these core services, there is also interest in smallest (or minimum cost) MCSes (i.e. the maximum satisfiability (MaxSAT) problem) and, in some settings, the smallest MUS problem (SMUS). It is well-known that the SMUS problem is hard for the second level of the polynomial hierarchy (Σ_2^{P}). Similarly, MCS and MUS membership problems are hard for Σ_2^{P}, and the specification of preferences of clauses to include in MUSes and MCSes are also hard for Σ_2^{P} [18].

Practical evidence suggests that complete enumeration of MUSes and MCSes is often beyond the reach of existing technologies, and a practical solution is the partial enumeration of either MUSes or MCSes [1,6,19,24,25,26]. A possible drawback of these approaches is that there is no guarantee that the actual sources of inconsistency in the given context are reported. An alternative solution is to compute the union of all

* This research is supported by the Spanish Government under project TIN2016-79190-R and by the Principality of Asturias under grant IDI/2018/000176. This work is also supported by FCT grants ABSOLV (PTDC/CCI-COM/28986/2017), FaultLocker (PTDC/CCI-COM/29300/2017), SAFETY (SFRH/BPD/120315/2016), and SAMPLE (CEECIND/04549/2017).

sources of inconsistency, i.e. the union of all MUSes (UMU). Information about UMU can restrict the number of clauses (or components in MBD) to analyze, with the guarantee that the source of inconsistency is present in the reported set.

Since MCS or MUS membership queries can be answered with a Σ_2^{P} oracle [10], the union of MUSes or MCSes can be obtained with a linear (but still large) number of calls to that oracle. However, it is often unrealistic in practice to call a Σ_2^{P} oracle say tens of thousands of times or more. Yet another approach consists in computing the intersection of all maximal satisfiable subsets (MSSes) and, from this compute the union of MUSes [14]. However, as hinted above, it is often unrealistic to enumerate all MCSes (and so MSSes). In contrast, this paper describes a novel algorithm for the direct computation of the union of MUSes. The experimental results provide evidence that the proposed algorithm offers a viable alternative to existing approaches.

2 Preliminaries

We consider propositional formulas in *conjunctive normal form* (CNF), defined as a conjunction, or set, of clauses $\mathcal{F} = \{c_1, c_2, ..., c_m\}$ over a set of variables $V(\mathcal{F}) = \{x_1, x_2, ..., x_n\}$, where a clause is a disjunction of literals, and a literal is a variable x or its negation $\neg x$.

A truth assignment, or interpretation, is a mapping $\mu : V(\mathcal{F}) \rightarrow \{0, 1\}$. If μ satisfies a formula $\mathcal{F}$, μ is referred to as a *model* of $\mathcal{F}$. Given two formulas $\mathcal{F}$ and $\mathcal{G}$, $\mathcal{F}$ entails $\mathcal{G}$ (written $\mathcal{F} \models \mathcal{G}$) iff all the models of $\mathcal{F}$ are models of $\mathcal{G}$. A formula $\mathcal{F}$ is satisfiable (written $\mathcal{F} \not\models \bot$) iff there exists a model for it, and unsatisfiable (written $\mathcal{F} \models \bot$) otherwise. SAT is the NP-complete problem [4] of deciding the satisfiability of a formula.

The following definitions characterize two dual notions in the analysis of unsatisfiable CNF formulas:

Definition 1 (MUS). *$\mathcal{M} \subseteq \mathcal{F}$ is a* minimally unsatisfiable subformula *(MUS) if and only if $\mathcal{M} \models \bot$ and for all $\mathcal{M}' \subsetneq \mathcal{M}$, $\mathcal{M}' \not\models \bot$.*

Definition 2 (MCS). *$\mathcal{C} \subseteq \mathcal{F}$ is a* minimal correction subset *(MCS) if and only if $(\mathcal{F} \setminus \mathcal{C}) \not\models \bot$ and for all $\mathcal{C}' \subsetneq \mathcal{C}$, $(\mathcal{F} \setminus \mathcal{C}') \models \bot$.*

MUSes represent minimal explanations of unsatisfiability, while MCSes represent irreducible subsets of clauses whose removal renders $\mathcal{F}$ satisfiable. Every MUS (resp. MCS) is a minimal hitting set of the set of all MCSes (resp. MUSes) [3,27]. There can be a worst-case exponential number of MUSes and MCSes [20].

Among the clauses in the MUSes of $\mathcal{F}$, a special case is that of *necessary* clauses. A clause c is necessary for (the unsatisfiability of) $\mathcal{F}$ iff $(\mathcal{F} \setminus \{c\}) \not\models \bot$. Necessary clauses belong to all the MUSes of a formula. In a broader sense, a clause is said to be *potentially necessary* [15] (*pn-clause* in short) iff it belongs to some MUS of $\mathcal{F}$, since it can become necessary after the removal of other clauses. The problem of deciding whether a given clause is potentially necessary is known to be Σ_2^{P}-complete (as the proof of Theorem 4 in [18] shows).

This paper addresses the computation of the set of all pn-clauses of an unsatisfiable formula $\mathcal{F}$, i.e., the union of its MUSes, denoted $\mathtt{UMU}(\mathcal{F})$. By the aforementioned hitting set duality relationship, the union of all MCSes of $\mathcal{F}$ is the same as $\mathtt{UMU}(\mathcal{F})$.

Input: $\mathcal{F}_0$ an unsatisfiable CNF formula
Output: $\texttt{CUMU}$, the union of MUSes of $\mathcal{F}_0$

1 $\mathcal{F}_0 \leftarrow \texttt{LeanKernel}(\mathcal{F}_0)$
2 $\mathcal{M} \leftarrow \texttt{ComputeMUS}(\mathcal{F}_0, \emptyset)$
3 $\texttt{CUMU} \leftarrow \mathcal{M}$
4 $\texttt{nec} \leftarrow \texttt{ComputeNecessaryClauses}(\mathcal{F}_0, \mathcal{M})$
5 **if** $\texttt{nec} \neq \mathcal{M}$ **then**
6 $\quad \mathcal{F}_0 \leftarrow \mathcal{F}_0 \setminus \{c \in (\mathcal{F}_0 \setminus \texttt{nec}); \texttt{nec} \models \{c\}\}$
7 $\quad \texttt{UMU\_rec}(\mathcal{F}_0, \texttt{nec}, \texttt{nec})$
8 **end**
9 **return** $\texttt{CUMU}$

Algorithm 1: Main procedure

A useful notion for *efficiently* over-approximating $\texttt{UMU}(\mathcal{F})$ is the *lean kernel* of $\mathcal{F}$, defined as the set of clauses used in some resolution refutation of $\mathcal{F}$, or, equivalently, those clauses which are not touched by any autarky (see [13] for the background, and see [16] for a recent paper on computing the lean kernel). A set of variables $A \subseteq V(\mathcal{F})$ is an *autarky* (or an *autarky varset*) iff there exists a truth assignment to the variables in A that satisfies all the clauses in $\mathcal{F}$ containing literals in the variables of A. There exists a unique largest autarky (varset), since the union of autarky (varsets) is again an autarky (varset). The lean kernel of $\mathcal{F}$ is obtained by removing the clauses containing some literal in the variables of the largest autarky of $\mathcal{F}$. In [15], the clauses in the lean kernel of $\mathcal{F}$ are referred to as *useful* clauses. Although some of these might not belong to $\texttt{UMU}(\mathcal{F})$, they can participate in short extended-resolution refutations of $\mathcal{F}$.

Example 1. Consider $\mathcal{F}_{ex} = \{c_1\colon (x_1), c_2\colon (\neg x_1), c_3\colon (\neg x_2), c_4\colon (\neg x_1 \vee x_2), c_5\colon (\neg x_2 \vee x_1), c_6\colon (\neg x_3 \vee x_4), c_7\colon (\neg x_4 \vee x_3)\}$. The largest autarky is $A = \{x_3, x_4\}$, so the lean kernel of $\mathcal{F}_{ex}$ is the clause-set $\{c_1, ..., c_5\}$. $\mathcal{F}_{ex}$ has the two MUSes $\mathcal{M}_1 = \{c_1, c_2\}$, $\mathcal{M}_2 = \{c_1, c_3, c_4\}$; and the three MCSes $\mathcal{C}_1 = \{c_1\}, \mathcal{C}_2 = \{c_2, c_3\}, \mathcal{C}_3 = \{c_2, c_4\}$. The union of MUSes is $\texttt{UMU}(\mathcal{F}_{ex}) = \{c_1, c_2, c_3, c_4\}$.

3 Computing the Union of MUSes

Consider an unsatisfiable formula $\mathcal{F}_0$. The main procedure for computing $\texttt{UMU}(\mathcal{F}_0)$ is shown in Algorithm 1. The core of our algorithm is recursive, with the initial recursive call in Line 7, and the recursive function $\texttt{UMU\_rec}$ given in Algorithm 2. Our approach develops a recursive enumeration of MUSes, keeping track of all the clauses found to be in $\texttt{UMU}(\mathcal{F}_0)$ along the search in the *global* under-approximation $\texttt{CUMU} \subseteq \texttt{UMU}(\mathcal{F}_0)$. Variable $\texttt{CUMU}$ is always a subset of $\texttt{UMU}(\mathcal{F}_0)$, and equality holds upon termination (which is guaranteed). Each node of the recursion tree is characterized by a subformula $\mathcal{F} \subseteq \mathcal{F}_0$, from which an MUS $\mathcal{M} \subseteq \mathcal{F}$ is extracted. The MUS $\mathcal{M}$ is then used for splitting, removing one clause $c_i \in \mathcal{M}$ from $\mathcal{F}$ in each new branch. So, along any path of the recursion tree, the current $\mathcal{F}$ is shrinking, by removing clauses, while $\texttt{CUMU} \subseteq \texttt{UMU}(\mathcal{F}_0)$ is growing, over the whole search. So at any point we can abort the computation (say, due to a time limit), and can use the current $\texttt{CUMU}$ as an under-approximation of $\texttt{UMU}(\mathcal{F}_0)$.

Global: CUMU, under-approximation of the union of MUSes
$\mathcal{F}_0$, initial formula (for early termination checks)
Input: $\mathcal{F}$, nec, forced, clause sets
Output: Boolean value, indicating if CUMU $= \mathcal{F}_0$

```
1  F ← LeanKernel(F)
2  if (F ⊆ CUMU) or (forced ⊈ F) then return false
3  M ← ComputeMUS(F, nec)
4  CUMU ← CUMU ∪ M
5  if CUMU = F_0 then return true
6  if (M = F) or (F ⊆ CUMU) then return false
7  for c ∈ (M \ forced) do
8      if SAT(F \ {c}) then
9          nec ← (nec ∪ {c})
10     else
11         if UMU_rec(F \ {c}, nec, forced) then return true
12         if F ⊆ CUMU then return false
13     end
14     forced ← (forced ∪ {c})
15     if not SAT(forced) then return false
16 end
17 return false
```

Algorithm 2: Recursive function UMU_rec

Some preprocessing steps are taken in Algorithm 1. In Line 1 the input formula $\mathcal{F}_0$ is reduced to its lean kernel, potentially removing a number of clauses not in $\mathtt{UMU}(\mathcal{F}_0)$. Then in Line 2 an MUS $\mathcal{M} \subseteq \mathcal{F}_0$ is extracted, which serves to initialize CUMU, and to compute the set nec of all necessary clauses of $\mathcal{F}_0$ in Line 4 (note $\mathtt{nec} \subseteq \mathcal{M}$). If $\mathtt{nec} = \mathcal{M}$ then the algorithm terminates, since $\mathcal{F}_0$ only contains one MUS (due to different MUSes being incomparable regarding set-inclusion). Otherwise, clauses in $\mathcal{F}_0 \setminus \mathtt{nec}$ entailed by nec are removed in Line 6 (no MUS of $\mathcal{F}_0$ contains such a clause, since it must contain nec as well), and the recursive procedure is invoked (parameters are passed by value). Note that CUMU can be initialized with better under-approximations of $\mathtt{UMU}(\mathcal{F}_0)$ than a single MUS (e.g. the union of some MCSes or MUSes known).

Now we come to UMU_rec (Algorithm 2). First we need to explain the structure of the splitting (the loop from Line 7 to Line 16). We split on any MUS $\mathcal{M} \subseteq \mathcal{F}$, computed in Line 3, where nec is now (only) some set of necessary clauses of $\mathcal{F}$, possibly missing some (only at the root we spend the effort to compute all). We add the clauses in $\mathcal{M}$ to CUMU (Line 4), and return (i.e., abort this branch of the recursion tree) if CUMU is all of the original $\mathcal{F}_0$ (Line 5; here indeed everything is finished), or when $\mathcal{F}$ cannot possibly contain any new clauses for CUMU (Line 6). Let $\mathcal{M} = \{c_1, \ldots, c_k\}$; the basic splitting idea is that branch i *excludes* clause c_i for any MUS considered in that branch — no MUS yielding some new clause, other than what we got from $\mathcal{M}$, can contain all of $\mathcal{M}$. Now we make the branches *possibly* disjoint ("morally" one might say), by *possibly including* in branch i the clauses $c_1, \ldots, c_{i-1}$. We note that the search is complete: Consider another MUS $\mathcal{M}'$ of $\mathcal{F}$ which contributes a new clause to CUMU. So there is some i with $c_i \notin \mathcal{M}'$, and for the smallest such i we have $c_1, \ldots, c_{i-1} \in \mathcal{M}'$.

The exclusion is lazily noted by adding the excluded clause to clause-set forced. We say "possibly exclude", which means the following: The branch can be aborted, if it is determined (by some incomplete, but correct argument — we prune "as good as we can") that every MUS $\mathcal{M} \subseteq \mathcal{F}$ with $\texttt{forced} \subset \mathcal{M}$ fulfils $\mathcal{M} \subseteq \texttt{CUMU}$. On the other side, we can include into CUMU in Line 4 whatever we want (as long as it belongs to $\texttt{UMU}(\mathcal{F}_0)$), the algorithm will always be correct.

The clause-set forced is the third argument of function UMU_rec, set in the original invocation to nec (Algorithm 1, Line 7), and updated in Line 14 of Algorithm 2. The second argument is the current value of nec, which is updated in Line 9: if $\mathcal{F}$ without c is satisfiable, then c is a necessary clause for $\mathcal{F}$, and considering the branch without c (Line 11 and Line 12) is useless.

It remains to discuss the remaining simplifications and shortcuts. In Line 1, $\mathcal{F}$ is reduced to its lean kernel (recall that no clause satisfied by some autarky can be element of any MUS). If $\mathcal{F}$ cannot contribute new clauses, or falls outside of forced, then we return (Line 2). The return value true of UMU_rec means that we are done completely, since all of $\mathcal{F}_0$ (the lean kernel of the original formula minus clauses entailed by the necessary clauses) has been covered by CUMU.

We always maintain the invariants $\texttt{nec} \subseteq \texttt{forced}$, and that forced is satisfiable (while irredundancy holds initially, but may be lost along a path — experimentation showed the irredundancy test to be rather expensive). We conclude by summarizing the argumentation of the section:

Theorem 1. *Algorithm 1 correctly computes* $\texttt{UMU}(\mathcal{F}_0)$ *and terminates for all inputs* $\mathcal{F}_0$.

3.1 Additional Procedures

Each node in the search tree defined by Algorithm 2 involves a number of computations requiring SAT oracles, besides the tests in Line 8 and Line 15.

The procedure $\texttt{ComputeMUS}(\mathcal{F}, \texttt{nec})$ computes an MUS $\mathcal{M} \subseteq \mathcal{F}$, where nec is a subset of the necessary clauses of $\mathcal{F}$. It follows a deletion-based approach with clause-set refinement [2]. In short, the procedure maintains two sets of clauses: $\mathcal{M}$, initialized with the clauses in nec, and $\mathcal{R}$, a *reference* set of clauses initially set to $\mathcal{F} \setminus \texttt{nec}$. As an invariant, $(\mathcal{M} \cup \mathcal{R}) \models \bot$. Iteratively, it tests the satisfiability of $(\mathcal{M} \cup \mathcal{R} \setminus \{c\})$, for some $c \in \mathcal{R}$. If satisfiable, c is moved to $\mathcal{M}$. Otherwise, $\mathcal{R}$ is updated to $\mathcal{C} \setminus \mathcal{M}$, with $\mathcal{C}$ an unsatisfiable core produced by the SAT solver. The procedure terminates when $\mathcal{R}$ becomes empty, returning the MUS $\mathcal{M}$, requiring $|\mathcal{F}| - |\texttt{nec}|$ satisfiability tests at most.

Given an MUS $\mathcal{M} \subseteq \mathcal{F}$, the procedure $\texttt{ComputeNecessaryClauses}(\mathcal{F}, \mathcal{M})$ identifies all the necessary clauses of $\mathcal{F}$. For each clause $c \in \mathcal{M}$ it tests the satisfiability of $\mathcal{F} \setminus \{c\}$. The clause c is necessary iff the latter formula is satisfiable. This procedure is only used in Algorithm 1 before the invocation to the recursive procedure, where (some) necessary clauses along a path are identified by a call to the SAT solver in the splitting step (Line 8 in Algorithm 2).

On the other hand, the procedure $\texttt{LeanKernel}(\mathcal{F})$ computes the lean kernel of $\mathcal{F}$ by first computing its largest autarky following the approach in [21]. Since this step is performed at every node (on different subsets of $\mathcal{F}_0$) we use a dedicated instantiation of a SAT solver for this task, which operates on an extension of the encoding Γ_3

proposed in [21] for representing largest autarkies. The encoding is extended with a selector variable s_i for each clause $c_i \in \mathcal{F}$, so that subformulas of $\mathcal{F}$ can be *activated* via assumptions, enabling reusing the encoding across the computation of the lean kernel of different subformulas. Initially, a formula referred to as $\mathcal{F}_{aut}$ is built from $\mathcal{F}_0$. Each variable $x_i \in V(\mathcal{F}_0)$ results in two variables $x_i^0, x_i^1 \in V(\mathcal{F}_{aut})$. Then, for each clause $c_i \in \mathcal{F}_0$ the following clauses are added, where $P(c_i)$ (resp. $N(c_i)$) denotes the positive (resp. negative) literals in c_i: (i) each $x_j \in P(c_i)$ results in the clause $s_i \rightarrow (x_j^0 \rightarrow \bigvee_{x_k \in P(c_i) \setminus \{x_j\}} x_k^1 \vee \bigvee_{x_k \in N(c_i)} x_k^0)$; (ii) each $x_j \in N(c_i)$ results in the clause $s_i \rightarrow (x_j^1 \rightarrow \bigvee_{x_k \in P(c_i)} x_k^1 \vee \bigvee_{x_k \in N(c_i) \setminus \{x_j\}} x_k^0)$. Finally, AtMost1 constraints of the form $(\neg x_i^0 \vee \neg x_i^1)$ for all $x_i \in V(\mathcal{F}_0)$ are added to $\mathcal{F}_{aut}$. The resulting encoding has $2n + m$ variables and $L + n$ clauses, where n is the number of variables, m is the number of clauses and L is the number of literals in $\mathcal{F}$.

Then, the computation of the largest autarky of any given $\mathcal{F} \subseteq \mathcal{F}_0$ amounts to computing a model of the formula $\mathcal{F}_{aut} \wedge \bigwedge_{\{c_i \in \mathcal{F}\}} s_i$ that maximizes the number of variables x_i^0 and x_i^1 set to true. This model is unique, so it corresponds to a maximal model (w.r.t. variables x_i^0 and x_i^1), that can be computed by reducing the problem to the extraction of an MCS of a formula considering the clauses in $\mathcal{F}_{aut} \wedge \bigwedge_{\{c_i \in \mathcal{F}\}} s_i$ as *hard* and a *soft* unit clause for each variable x_i^0 and x_i^1. For this purpose, we use the *Clause D* algorithm, as proposed in [21]. The set of variables appearing in the complement of the computed MCS corresponds to the largest autarky of $\mathcal{F}$. The lean kernel of $\mathcal{F}$ is obtained by removing all the clauses containing a variable in the computed autarky.

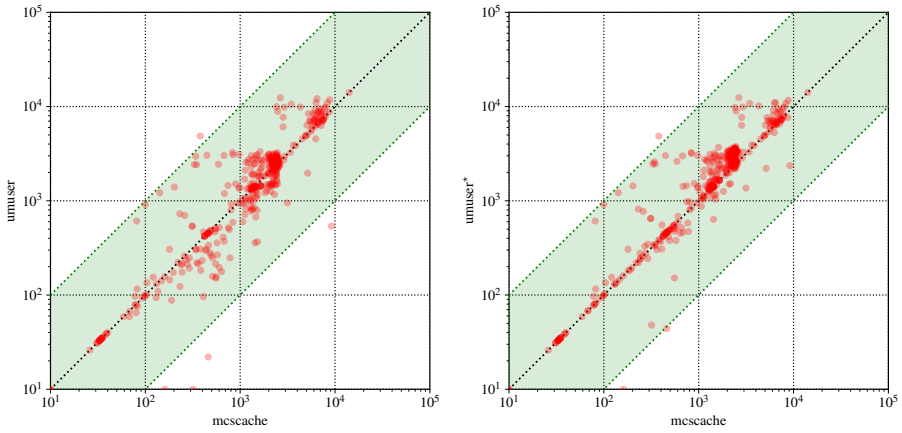
4 Experimental Results

This section presents an experimental assessment of the proposed approach to computing the union of MUSes, $\texttt{UMU}(\mathcal{F})$, of an unsatisfiable formula $\mathcal{F}$. The experiments were performed on a Linux machine (Intel Xeon 2.26GHz, 128GByte). The time limit for each process was set to 900s and the memory limit to 4GByte.

A prototype implementing the proposed algorithm was written in C++ on top of the known *caching* MCS enumerator *mcscache* proposed in [25]. In the following, the prototype is referred to as *umuser*. Among the existing alternative state-of-the-art MCS and MUS enumerators [6,24,25,26], we opted to compare the prototype against *mcscache* as they share the same code base and the interface to a SAT solver. The underlying SAT solver used in both tools is MiniSat 2.2 [5]. Additionally and following the ideas of Section 3, we considered a combination of both tools connected in the following setup: (i) first, *mcscache* was used to enumerate MCSes within 3 minutes and (ii) second, *umuser* was bootstrapped with the clauses in these MCSes as an initial approximation of $\texttt{UMU}(\mathcal{F})$. This approach is referred to as *umuser**.

The considered benchmark suite includes two sets of instances. The first one is derived from the MUS track of the SAT competition 2011[5]. These benchmarks were widely used in prior work on MUS and MCS computation and enumeration [2,19,22,23]. Since computing all clauses participating in an MUS/MCS is computationally hard, for our evaluation purposes, we took only instances with at most 20000 clauses from this

[5] https://satcompetition.org/2011/



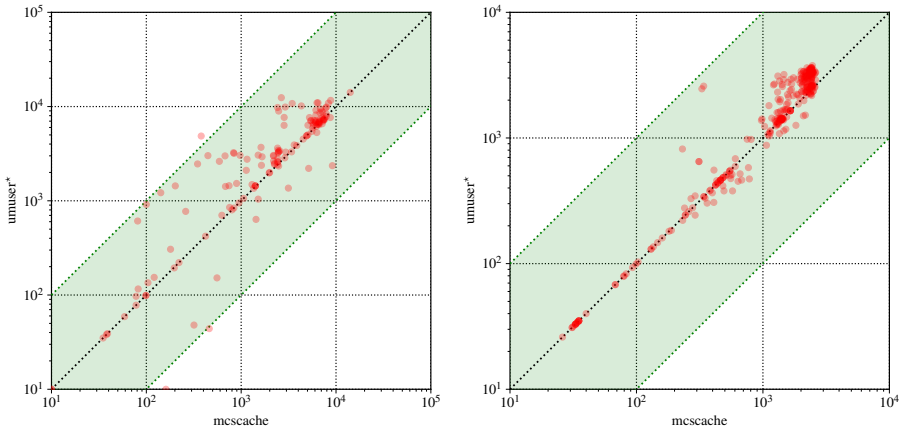
(a) *umuser* vs. *mcscache* (b) *umuser** vs. *mcscache*

Fig. 1: The number of clauses in $\texttt{UMU}(\mathcal{F})$ computed within 900 seconds.

benchmark set. The second set of instances comprises the benchmarks previously studied in a number of settings, including MCS [22] and SMUS (smallest MUS) [7] computation. These include automotive product configuration benchmarks [30] and circuit diagnosis. The total number of benchmark instances considered in both sets is 427.

First of all, it should be noted that *mcscache* terminates for 98 instances while the default setup of *umuser* can finish only for 41 of them by the time limit. This does not come as a surprise provided that the recursive procedure of *umuser* is computationally more expensive than simple MCS enumeration. In general, when targeting instances with few MCSes, an MCS enumerator may be the best approach to use, while *umuser* should be reasonable to apply when exhaustive enumeration of MCSes or MUSes is infeasible. Also note that the combined variant *umuser** terminates for 94 benchmark instances, several of them solved during the MCS enumeration phase. Similar results were observed with a longer time limit of 30 min, with only 7 more instances solved.

To illustrate the power of the proposed approach w.r.t. under-approximations of $\texttt{UMU}(\mathcal{F})$, the second and the most important part of the experiment is to evaluate the number of clauses reported to participate in $\texttt{UMU}(\mathcal{F})$. Figure 1 shows the comparison between *umuser*/*umuser** and *mcscache* in terms of the number of clauses in $\texttt{UMU}(\mathcal{F})$ found within 900 seconds. As can be seen in Figure 1a, apart from a number of outliers that can be exhaustively solved by *mcscache*, the basic version of *umuser* tends to compute more clauses than what can be achieved by *mcscache*, indicating its effectiveness at conducting the search towards new clauses in $\texttt{UMU}(\mathcal{F})$. Furthermore, as shown in Figure 1b, *umuser** strengthens the approach more, which enables it to find even more clauses. Concretely, the average number of clauses in $\texttt{UMU}(\mathcal{F})$ per instance computed by *mcscache* within 900 seconds is 2089.58, while *umuser* and *umuser** compute 2421.87 and 2623.14, respectively. A more detailed comparison between *umuser** and *mcscache* is shown in Figure 2, where the two considered benchmark sets are analyzed separately.



(a) Benchmark set 1 (b) Benchmark set 2

Fig. 2: The number of clauses in $\texttt{UMU}(\mathcal{F})$ computed for the two benchmark sets.

Both Figure 2a and Figure 2b confirm that *umuser** has an advantage over *mcscache* in terms of the number of clauses identified to belong to $\texttt{UMU}(\mathcal{F})$.

Computing lean kernels allows the identification of clauses not in $\texttt{UMU}(\mathcal{F})$. At the root of the recursion tree, on average 242 such clauses were identified, with an average reduction of 15.9% w.r.t. the number of clauses of the formula. The extent of this reduction depends on each instance, ranging from 0% (62 instances) to 97.32%, and a median value of 3.32%. These results encourage future research on strategies for activating and deactivating the computation of lean kernels on demand, depending on its effectiveness.

5 Conclusions

Identifying clauses that take part in the unsatisfiability of a formula represents a useful task in infeasibility analysis. In this context, clauses might be classified in three levels regarding their *importance*. The extreme cases are the set of necessary clauses, which belong to all MUSes, and the lean kernel, containing all the clauses that can be used in a resolution refutation. In the middle, the union of MUSes, $\texttt{UMU}(\mathcal{F})$, constitutes a concise (and accurate) explanation of all the causes of unsatisfiability. This paper addresses the Σ_2^{P}-hard problem of computing $\texttt{UMU}(\mathcal{F})$ and proposes a novel algorithm for this task. Based on a lazy splitting along MUSes (similar to splitting on a clause in SAT solving) and a global collection of clauses found for $\texttt{UMU}(\mathcal{F})$, the algorithm uses a variety of NP oracles for powerful pruning techniques, obtaining a shortened implicit enumeration of MUSes. These pruning steps can be relaxed or strengthened in various ways, giving rise to a general framework for computing $\texttt{UMU}(\mathcal{F})$, and opening a wide space of promising possibilities for the future. Experimental results show that under a given time limit the new algorithm produces (much) better approximations of $\texttt{UMU}(\mathcal{F})$ than an approach based on MCS enumeration. The results also reveal potential benefits of combining both approaches, encouraging further research on alternative methods.

References

1. Bacchus, F., Katsirelos, G.: Finding a collection of MUSes incrementally. In: Quimper, C. (ed.) Integration of AI and OR Techniques in Constraint Programming - 13th International Conference, CPAIOR 2016, Banff, AB, Canada, May 29 - June 1, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9676, pp. 35–44. Springer (2016). https://doi.org/10.1007/978-3-319-33954-2_3
2. Belov, A., Lynce, I., Marques-Silva, J.: Towards efficient MUS extraction. AI Commun. **25**(2), 97–116 (2012). https://doi.org/10.3233/AIC-2012-0523
3. Birnbaum, E., Lozinskii, E.L.: Consistent subsets of inconsistent systems: structure and behaviour. J. Exp. Theor. Artif. Intell. **15**(1), 25–46 (2003). https://doi.org/10.1080/0952813021000026795
4. Cook, S.A.: The complexity of theorem-proving procedures. In: Harrison, M.A., Banerji, R.B., Ullman, J.D. (eds.) Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA. pp. 151–158. ACM (1971). https://doi.org/10.1145/800157.805047
5. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers. Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer (2003). https://doi.org/10.1007/978-3-540-24605-3_37
6. Grégoire, É., Izza, Y., Lagniez, J.: Boosting MCSes enumeration. In: Lang [17], pp. 1309–1315. https://doi.org/10.24963/ijcai.2018/182
7. Ignatiev, A., Janota, M., Marques-Silva, J.: Quantified maximum satisfiability. Constraints **21**(2), 277–302 (2016). https://doi.org/10.1007/s10601-015-9195-9
8. Jannach, D., Schmitz, T.: Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach. Autom. Softw. Eng. **23**(1), 105–144 (2016). https://doi.org/10.1007/s10515-014-0141-7
9. Jannach, D., Schmitz, T., Hofer, B., Schekotihin, K., Koch, P.W., Wotawa, F.: Fragment-based spreadsheet debugging. Autom. Softw. Eng. **26**(1), 203–239 (2019). https://doi.org/10.1007/s10515-018-0250-9
10. Janota, M., Marques-Silva, J.: On deciding MUS membership with QBF. In: Lee, J.H. (ed.) Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6876, pp. 414–428. Springer (2011). https://doi.org/10.1007/978-3-642-23786-7_32
11. Jose, M., Majumdar, R.: Bug-assist: Assisting fault localization in ANSI-C programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 504–509. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_40
12. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: Hall, M.W., Padua, D.A. (eds.) Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011. pp. 437–446. ACM (2011). https://doi.org/10.1145/1993498.1993550
13. Kleine Büning, H., Kullmann, O.: Minimal unsatisfiability and autarkies. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 339–401. IOS Press (2009). https://doi.org/10.3233/978-1-58603-929-5-339

14. Kullmann, O.: An application of matroid theory to the SAT problem. In: Proceedings of the 15th Annual IEEE Conference on Computational Complexity, Florence, Italy, July 4-7, 2000. p. 116. IEEE Computer Society (2000). https://doi.org/10.1109/CCC.2000.856741
15. Kullmann, O., Lynce, I., Marques-Silva, J.: Categorisation of clauses in conjunctive normal forms: Minimally unsatisfiable sub-clause-sets and the lean kernel. In: Biere, A., Gomes, C.P. (eds.) Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4121, pp. 22–35. Springer (2006). https://doi.org/10.1007/11814948_4
16. Kullmann, O., Marques-Silva, J.: Computing maximal autarkies with few and simple oracle queries. In: Heule, M., Weaver, S. (eds.) Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9340, pp. 138–155. Springer (2015). https://doi.org/10.1007/978-3-319-24318-4_11
17. Lang, J. (ed.): Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden. ijcai.org (2018)
18. Liberatore, P.: Redundancy in logic I: CNF propositional formulae. Artif. Intell. **163**(2), 203–232 (2005). https://doi.org/10.1016/j.artint.2004.11.002
19. Liffiton, M.H., Previti, A., Malik, A., Marques-Silva, J.: Fast, flexible MUS enumeration. Constraints **21**(2), 223–250 (2016). https://doi.org/10.1007/s10601-015-9183-0
20. Liffiton, M.H., Sakallah, K.A.: Searching for autarkies to trim unsatisfiable clause sets. In: Büning, H.K., Zhao, X. (eds.) Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4996, pp. 182–195. Springer (2008). https://doi.org/10.1007/978-3-540-79719-7_18
21. Marques-Silva, J., Ignatiev, A., Morgado, A., Manquinho, V.M., Lynce, I.: Efficient autarkies. In: Schaub, T., Friedrich, G., O'Sullivan, B. (eds.) ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014). Frontiers in Artificial Intelligence and Applications, vol. 263, pp. 603–608. IOS Press (2014). https://doi.org/10.3233/978-1-61499-419-0-603
22. Mencía, C., Ignatiev, A., Previti, A., Marques-Silva, J.: MCS extraction with sublinear oracle queries. In: Creignou, N., Berre, D.L. (eds.) Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9710, pp. 342–360. Springer (2016). https://doi.org/10.1007/978-3-319-40970-2_21
23. Mencía, C., Previti, A., Marques-Silva, J.: Literal-based MCS extraction. In: Yang, Q., Wooldridge, M.J. (eds.) Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015. pp. 1973–1979. AAAI Press (2015), http://ijcai.org/Abstract/15/280
24. Narodytska, N., Bjørner, N., Marinescu, M., Sagiv, M.: Core-guided minimal correction set and core enumeration. In: Lang [17], pp. 1353–1361. https://doi.org/10.24963/ijcai.2018/188
25. Previti, A., Mencía, C., Järvisalo, M., Marques-Silva, J.: Improving MCS enumeration via caching. In: Gaspers, S., Walsh, T. (eds.) Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10491, pp. 184–194. Springer (2017). https://doi.org/10.1007/978-3-319-66263-3_12
26. Previti, A., Mencía, C., Järvisalo, M., Marques-Silva, J.: Premise set caching for enumerating minimal correction subsets. In: McIlraith, S.A., Weinberger, K.Q. (eds.) Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on

Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018. pp. 6633–6640. AAAI Press (2018), https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17328
27. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. **32**(1), 57–95 (1987). https://doi.org/10.1016/0004-3702(87)90062-2
28. Safarpour, S., Mangassarian, H., Veneris, A.G., Liffiton, M.H., Sakallah, K.A.: Improved design debugging using maximum satisfiability. In: Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings. pp. 13–19. IEEE Computer Society (2007). https://doi.org/10.1109/FAMCAD.2007.26
29. Schlobach, S., Huang, Z., Cornet, R., van Harmelen, F.: Debugging incoherent terminologies. J. Autom. Reasoning **39**(3), 317–349 (2007). https://doi.org/10.1007/s10817-007-9076-z
30. Sinz, C., Kaiser, A., Küchlin, W.: Formal methods for the validation of automotive product configuration data. AI EDAM **17**(1), 75–97 (2003). https://doi.org/10.1017/S0890060403171065
31. Smith, A., Veneris, A.G., Ali, M.F., Viglas, A.: Fault diagnosis and logic debugging using Boolean satisfiability. IEEE Trans. on CAD of Integrated Circuits and Systems **24**(10), 1606–1621 (2005). https://doi.org/10.1109/TCAD.2005.852031
32. Stuckey, P.J., Sulzmann, M., Wazny, J.: Interactive type debugging in Haskell. In: Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2003, Uppsala, Sweden, August 28, 2003. pp. 72–83. ACM (2003). https://doi.org/10.1145/871895.871903