On Improving MUS Extraction Algorithms

Joao Marques-Silva[1,2] and Ines Lynce[2]

[1] University College Dublin jpms@ucd.ie
[2] INESC-ID/IST, TU Lisbon ines@sat.inesc-id.pt

Abstract. Minimally Unsatisfiable Subformulas (MUS) find a wide range of practical applications, including product configuration, knowledge-based validation, and hardware and software design and verification. MUSes also find application in recent Maximum Satisfiability algorithms and in CNF formula redundancy removal. Besides direct applications in Propositional Logic, algorithms for MUS extraction have been applied to more expressive logics. This paper proposes two algorithms for MUS extraction. The first algorithm is optimal in its class, meaning that it requires the smallest number of calls to a SAT solver. The second algorithm extends earlier work, but implements a number of new techniques. The resulting algorithms achieve significant performance gains with respect to state of the art MUS extraction algorithms.

1 Introduction

There has been a remarkable amount of recent work on algorithms for computing minimal explanations of unsatisfiability over the last decade (e.g. [28, 16, 3, 15, 14, 9–11, 27, 12, 7, 13, 23, 25]). Most of this work is inspired by earlier work on computing explanations for inconsistencies (e.g. [5, 4, 1]). Algorithms for MUS extraction have often been characterized as *constructive* [12] (also referred to as insertion-based [7, 23]), as *destructive* [12] (also referred to as removal-based [7], or deletion-based [23]), or as *dichotomic* [16, 14]. All MUS extraction algorithms involve a number of calls to a SAT solver (or some other NP oracle). For destructive approaches, the best performing algorithms require $\mathcal{O}(m)$ calls to a SAT solver, where m is the number of clauses in the original formula. Existing constructive approaches require $\mathcal{O}(m \times k)$ calls to a SAT solver, where k is the size of the largest MUS in the original CNF formula [12]. Finally, the dichotomic approach requires $\mathcal{O}(k \log m)$ calls to a SAT solver. Recent work proposed an approach based on a weighted Maximum Satisfiability (MaxSAT) solver [7], but the function problem associated with computing a weighted MaxSAT solution is in Δ_2^P, and so unlikely to be in NP. There is also a large body of work on computing *good* approximations of MUSes (e.g. [23]). Despite the large body of work, MUS extraction algorithms are *not* industrial-strength, meaning that, with a few recent exceptions (e.g. [25]), MUS extraction algorithms are seldom evaluated on large problem instances or used in practical settings. This is demonstrated in the results section of this paper, where existing MUS extraction algorithms are shown to be in general inefficient for large complex problem instances from practical applications.

This paper represents a first effort towards developing industrial-strength MUS extraction algorithms, and has the following main contributions. First, the paper develops a constructive algorithm for MUS extraction that requires $\mathcal{O}(m)$ calls to a SAT solver. This result implies (i) that destructive and constructive approaches have the same worst-case complexity in terms of the number of calls to a SAT solver; and (ii) that when $k = \Theta(m)$, the new algorithm represents the optimal case (as does the destructive algorithm). More importantly, this new algorithm blurs the distinction between destructive and constructive algorithms. Motivated by this observation, the paper proposes a hybrid algorithm that formally operates as a constructive algorithm, but that essentially exploits all steps of the algorithm to reduce the number of required iterations. This causes the algorithm to operate in a mostly hybrid mode, iteratively constructing the MUS, but also exploiting available information to reduce the number of iterations. Another contribution of the paper is the integration of a number of techniques that serve to simplify each SAT solver call, and to reduce the set of clauses that need to be analyzed through a call to a SAT solver. Moreover, the paper also shows that some existing techniques need not be considered for MUS extraction. Finally, the paper conducts a comprehensive evaluation of existing publicly available MUS extractors on representative industrial problem instances, obtained from well-known practical applications of SAT, where MUS extraction finds application.

2 Preliminaries

A set of variables $X = \{x_1, \ldots, x_N\}$ is assumed. A formula $\mathcal{F}$ in Conjunctive Normal Form (CNF) is defined as a set of sets of literals defined on X. A literal is either a variable or its complement. Each set of literals is referred to as a clause. Moreover, it is assumed that each clause is non-tautological. Given a clause c_i, $\{\neg c_i\}$ denotes the set of unit clauses obtained from negating c_i. Additional standard definitions can be found elsewhere (e.g. [8, 24]). The focus of this paper are unsatisfiable formulas, and the characterization of the sources of unsatisfiability. Throughout the paper, $\mathcal{F}$, $\mathcal{F}' \subseteq \mathcal{F}$, $\mathcal{F}^R$, $\mathcal{F}^I$ and $\mathcal{U}$ denote CNF formulas, $\mathcal{S}$ and $\mathcal{S}'$ denote MUSes of $\mathcal{F}$, and $\mathcal{M}$ denotes a subset of an MUS $\mathcal{S}$.

Definition 1 (MUS). *$\mathcal{M} \subseteq \mathcal{F}$ is a* Minimally Unsatisfiable Subset *(MUS) iff $\mathcal{M}$ is unsatisfiable and $\forall_{c \in \mathcal{M}}, \mathcal{M} \setminus \{c\}$ is satisfiable.*

Definition 2 (MCS). *$\mathcal{C} \subseteq \mathcal{F}$ is a* Minimal Correction Subset *(MCS) iff $\mathcal{F} \setminus \mathcal{C}$ is satisfiable and $\forall_{c \in \mathcal{C}}, \mathcal{F} \setminus (\mathcal{C} \setminus \{c\})$ is unsatisfiable.*

Throughout the paper, m denotes the number of clauses in the original CNF formula $\mathcal{F}$, $m = |\mathcal{F}|$, and k denotes the number of clauses in the largest MUS $\mathcal{M}$, $k = |\mathcal{M}|$. The MUS decision problem, i.e. the problem of *deciding* whether a CNF formula $\mathcal{F}$ is an MUS is D^P-complete. In contrast, the problem of *computing* an MUS from an unsatisfiable CNF formula requires a number of calls to a SAT oracle. Over the years, three main approaches have been proposed for computing an MUS: *constructive* [5], *destructive* [4, 1] and *dichotomic* [16, 14]. Constructive approaches require $\mathcal{O}(m \times k)$ calls to an NP-oracle, destructive approaches require $\mathcal{O}(m)$ calls, and dichotomic approaches require $\mathcal{O}(k \times \log m)$ calls. Despite the theoretical interest of the dichotomic algorithm, the most recent implementation of MUS extraction algorithms are either destructive [2, 25] or constructive [27].

Algorithm 1: Destructive MUS Extraction

```
Input  : Unsatisfiable CNF Formula $\mathcal{F}$
Output: MUS $\mathcal{M}$
1 begin
2     $\mathcal{M} \leftarrow \mathcal{F}$                              // MUS over-approximation
3     foreach $c_i \in \mathcal{M}$ do
4         if not SAT($\mathcal{M} \setminus \{c_i\}$) then          // $c_i$ is not transition clause
5             $\mathcal{M} \leftarrow \mathcal{M} \setminus \{c_i\}$
6     return $\mathcal{M}$                                   // Final $\mathcal{M}$ is an MUS
7 end
```

Most practical MUS computation algorithms iteratively identify *transition clauses* [12]. The following definition is used throughout this paper.

Definition 3 (Transition Clause). *Let $\mathcal{F}$ be an unsatisfiable set of clauses and let $c \in \mathcal{F}$ be a clause. If $\mathcal{F} \setminus \{c\}$ is satisfiable then c is a transition clause with respect to $\mathcal{F}$.*

Lemma 1. *Let c be a transition clause of CNF formula $\mathcal{F}$. Then c is included in* any *MUS of $\mathcal{F}$.*

Proof. $\mathcal{F} \setminus \{c\}$ is satisfiable. Any unsatisfiable subset of $\mathcal{F}$ must include c. □

Throughout the paper, SAT solvers are used as NP-oracles, that test the satisfiability of CNF formulas. In general, SAT($\mathcal{F}$) tests the satisfiability of a formula $\mathcal{F}$; it returns value true if the formula is satisfiable, and value false if the formula is unsatisfiable. Where necessary, SAT($\mathcal{F}$) may also return the satisfying assignment and an unsatisfiable subset. In this case, the output of the SAT solver call is represented as follows: $(\mathsf{st}, \nu, \mathcal{U}) \leftarrow \mathtt{SAT}(\mathcal{F})$. st is a Boolean variable assigned value *true* if the instance is satisfiable, in which case ν contains a solution to $\mathcal{F}$, or assigned value *false*, in which case $\mathcal{U} \subseteq \mathcal{F}$ is an unsatisfiable subformula. Besides the use of SAT solvers as NP-oracles, some algorithms propose the use of weighted MaxSAT solvers [7].

The standard organization of a destructive MUS extraction algorithm is shown in Algorithm 1 [12, 23]. The algorithm starts with a working formula $\mathcal{M}$ equal to the original formula $\mathcal{F}$. Iteratively, the algorithm checks whether each one of the clauses $c_i \in \mathcal{M}$ is a transition clause. Non transition clauses are removed from $\mathcal{M}$. In the end, $\mathcal{M}$ is an MUS. This algorithm is studied in more detail in later sections.

Recent overviews of MUS extraction algorithms can be found in [12, 7, 23].

3 New Constructive Algorithm for MUS Extraction

This section develops a new constructive algorithm, that takes $\mathcal{O}(m)$ calls to a SAT oracle. This result implies that constructive and destructive approaches for MUS extraction have the same worst-case complexity in terms of the number of calls to a SAT solver, and improves known results in this area [12, 23].

Algorithm 2 shows the new constructive MUS extraction algorithm. This new algorithm borrows ideas from a number of earlier algorithms. Similarly to AMUSE [26], it adds relaxation variables to all clauses. In addition, and similarly to the use of weighted

Algorithm 2: Constructive MUS Extraction with AtMost1 Constraint

```
Input  : Unsatisfiable CNF Formula $\mathcal{F}$
Output: MUS $\mathcal{M}$
begin
    $\mathcal{M} \leftarrow \emptyset$                                              // $\mathcal{M}$: MUS under-approximation
    $\mathcal{R} \leftarrow \{r_i \,|\, r_i$ is fresh variable for $c_i \in \mathcal{F}\}$        // $\mathcal{R}$: relaxation variables
    $\mathcal{F}^R \leftarrow \{c_i \cup \{r_i\} \,|\, r_i \in R \wedge c_i \in \mathcal{F}\}$           // $\mathcal{F}^R$: working formula
    $\mathcal{T} \leftarrow$ CNF($\sum_{r_i \in R} r_i \leq 1$)                         // $\leq 1$ constraint
    while $\mathcal{F}^R \neq \emptyset$ do                       // Repeat while relaxed clauses exist
        (st, $\nu$, $\mathcal{U}$) $\leftarrow$ SAT($\mathcal{F}^R \cup \mathcal{T} \cup \mathcal{M}$)
        if st = true then
            $r_i \leftarrow$ TrueVariable($\nu, R$)          // Get true relaxation variable
            $c_i^R \leftarrow$ Clause($\mathcal{F}^R, r_i$)          // Get clause associated with $r_i$
            $\mathcal{F}^R \leftarrow \mathcal{F}^R \setminus \{c_i^R\}$        // Remove clause $c_i^R = c_i \cup \{r_i\}$ from $\mathcal{F}^R$
            $\mathcal{M} \leftarrow \mathcal{M} \cup \{c_i^R \setminus \{r_i\}\}$        // Add clause $c_i = c_i^R \setminus \{r_i\}$ to MUS
        else                                          // If unsatisfiable, $\mathcal{U} \cap \mathcal{T} \neq \emptyset$
            if $\mathcal{U} \cap \mathcal{F}^R = \emptyset$ then
                $\mathcal{F}^R \leftarrow \emptyset$
            else
                $c_i^R \leftarrow$ SelectClause($\mathcal{F}^R \cap \mathcal{U}$)
                $\mathcal{F}^R \leftarrow \mathcal{F}^R \setminus \{c_i^R\}$                       // Block one MUS
    return $\mathcal{M}$                                        // Final $\mathcal{M}$ is an MUS
end
```

MaxSAT for MUS extraction [7], a SAT (resp. weighted MaxSAT) test is used to decide which clause to add to the MUS being built.

The operation of the algorithm is as follows. Assume the original formula $\mathcal{F}$ is unsatisfiable. The algorithm starts by creating a working formula $\mathcal{F}^R$ by relaxing all clauses in $\mathcal{F}$. An *AtMost1* constraint is created and encoded into the CNF formula $\mathcal{T}$, requiring at most one relaxation variable r_i to be assigned value true. $\mathcal{M}$ is initially an empty set and in the end is an MUS.

The outcome of the SAT solver call (see line 7) given formula $\mathcal{F}^R \cup \mathcal{T} \cup \mathcal{M}$ can either be true or false. If the outcome st is true, this means that exactly one relaxation variable was set to true. This relaxation variable r_i is associated with a clause c_i that is part of the MUS $\mathcal{M}$ being constructed. If st is false, this means that more than one relaxation variable would have to be assigned value true for the outcome to be true. This also implies the existence of more than one MUS, and so the solution is to (arbitrarily) block one MUS. This is done by simply removing a clause c_i^R from $\mathcal{F}^R$ that also occurs in the unsatisfiable formula $\mathcal{U}$ computed by the SAT solver. The process is iterated until $\mathcal{F}^R$ becomes empty (denoting that $\mathcal{M}$ is unsatisfiable), in which case $\mathcal{M}$ is an MUS.

To prove that Algorithm 2 computes an MUS of $\mathcal{F}$, the following intermediate results will be used.

Definition 4. *Throughout the execution of Algorithm 2, let $\mathcal{F}^I$ represent the clauses in $\mathcal{F}^R$ without the corresponding relaxation variables. (Observe that $\mathcal{F}^I \cap \mathcal{M} = \emptyset$.)*

Lemma 2. *Assume $\mathcal{M} \subsetneq \mathcal{S} \subseteq \mathcal{F}^I \cup \mathcal{M}$, where $\mathcal{S}$ is an MUS. Let $\mathcal{F}^R \cup \mathcal{T} \cup \mathcal{M}$ be unsatisfiable. Then $\mathcal{M}$ can be extended to strictly more than one MUS.*

Proof. Suppose that $\mathcal{M}$ can be extended to *exactly one* MUS $\mathcal{S}$. Select a clause c_i in $\mathcal{S} \setminus \mathcal{M}$, and relax clause c_i. By definition of MUS, $\mathcal{S} \setminus \{c_i\}$ must be satisfiable, and since $\mathcal{M}$ can be extended to exactly one MUS, then $\mathcal{F}^R \cup \mathcal{T} \cup \mathcal{M}$ would have to be satisfiable; a contradiction. □

Corollary 1. *Assume $\mathcal{M} \subsetneq \mathcal{S} \subseteq \mathcal{F}^I \cup \mathcal{M}$, where $\mathcal{S}$ is an MUS. Let $\mathcal{F}^R \cup \mathcal{T} \cup \mathcal{M}$ be unsatisfiable (i.e. line 13 of the algorithm), let $\mathcal{U}$ be an unsatisfiable subformula computed by the SAT solver, and let $(c_i \cup \{r_i\}) \in \mathcal{F}^R \cap \mathcal{U}$. Then there exists an MUS $\mathcal{S}'$ with $\mathcal{S}' \subseteq \mathcal{M} \cup (\mathcal{F}^I \setminus \{c_i\})$.*

Proof. $\mathcal{M} \cup (\mathcal{F}^R \setminus \{c_i \cup \{r_i\}\}) \cup \mathcal{T}$ is either satisfiable, requiring exactly one clause in $\mathcal{F}^R$ to be relaxed, or remains unsatisfiable. In either case, it still contains an MUS. □

Lemma 3. *Assume $\mathcal{M} \subsetneq \mathcal{S} \subseteq \mathcal{F}^I \cup \mathcal{M}$, where $\mathcal{S}$ is a MUS. Let $\mathcal{F}^R \cup \mathcal{T} \cup \mathcal{M}$ be satisfiable, and let c_i be a clause with an associated true relaxation variable r_i. Then, any MUS with clauses in $\mathcal{F}^I \cup \mathcal{M}$ will include c_i.*

Proof. By hypothesis, $\mathcal{F}^I \cup \mathcal{M}$ is unsatisfiable. If $\mathcal{F}^R \cup \mathcal{T} \cup \mathcal{M}$ is satisfiable, then $\mathcal{F}^R \cup \mathcal{M}$ has an MCS of size 1, which is identified by the relaxed clause c_i. Hence, by definition of MCS, c_i must be part of any MUS in $\mathcal{F}^I \cup \mathcal{M}$. □

Theorem 1. *Algorithm 2 returns an MUS of unsatisfiable CNF formula $\mathcal{F}$.*

Proof. To prove that Algorithm 2 computes on MUS of $\mathcal{F}$, the following invariants hold after each iteration of the algorithm: (i) $\mathcal{F}^I \cup \mathcal{M}$ is unsatisfiable; and (ii) there exists an MUS $\mathcal{S}$, with $\mathcal{M} \subseteq \mathcal{S} \subseteq \mathcal{F}^I \cup \mathcal{M}$. The invariants can be proved by induction on the number of iterations of the algorithm. Clearly, the invariants hold for the base case, with $\mathcal{M} = \emptyset$ and $\mathcal{F}^I$ unsatisfiable. Suppose that the invariants hold after iteration $j - 1$. Then, the objective is to analyze the invariants after iteration j. Suppose the SAT call in line 7 returns false. Hence, one clause is removed from $\mathcal{F}^I$. From Lemma 2 and Corollary 1, it is guaranteed that the resulting formula $\mathcal{F}^I \cup \mathcal{M}$ is still unsatisfiable and contains an MUS. Alternatively, suppose the SAT call in line 7 returns true. Hence, the relaxation variable is removed from the identified relaxed clause and the clause is added to $\mathcal{M}$. From Lemma 3, the identified clause is included in any MUS, and so can be added to $\mathcal{M}$. Moreover, the two invariants still hold: $\mathcal{M}$ continues to be part of an MUS and $\mathcal{F}^I \cup \mathcal{M}$ is unsatisfiable. □

Lemma 4. *The number of calls to a SAT solver by Algorithm 2 is in $\Theta(m)$.*

Proof. To prove that the number of calls is $\mathcal{O}(m)$, observe that the algorithm removes one clause from $\mathcal{F}^R$ at each iteration of the loop. Hence, there can be at most m calls to a SAT solver. To prove that the number of calls is $\Omega(m)$, consider the following CNF formula $\mathcal{F} = \{\neg x_1\} \cup_{i=1}^{N-1} \{x_i, \neg x_{i+1}\} \cup \{x_N\}$, with $|\mathcal{F}| = N + 1 = m$. $\mathcal{F}$ has a single MUS, containing all clauses. Each iteration of the algorithm will add exactly one clause to $\mathcal{M}$. Hence, the number of calls to the SAT solver is $N + 1 = m$. Thus, the number of calls to a SAT solver is in $\Omega(m)$. □

Lemma 4 shows that deletion-based and insertion-based MUS extraction algorithms can have the same asymptotic complexity in terms of the number of calls to a SAT solver. Moreover, Algorithm 2 provides one concrete example of such algorithm. It should be noted that Algorithm 2 runs the SAT solver on a modified problem instance. However, as will be shown later, despite working on a modified problem instance, Algorithm 2 provides a few practical advantages.

4 Hybrid MUS Extraction

One of the interesting aspects of Algorithm 2 is that it blurs the distinction between constructive and destructive algorithms. On the one hand, the algorithm iteratively expands a subset of an MUS. On the other hand, the algorithm requires $\mathcal{O}(m)$ calls to a SAT solver. Similarly, one can develop a variant of Algorithm 1 that is essentially a constructive algorithm. Algorithm 3 shows this variant. As with Algorithm 2, $\mathcal{M}$ denotes a subset of an MUS, and the number of calls to a SAT solver is $\mathcal{O}(m)$. Nevertheless, Algorithm 3 also shares similarities with Algorithm 1, namely that each clause is analyzed exactly once, thus guaranteeing $\Theta(m)$ calls to a SAT solver. Besides the minor changes needed to make a constructive variant of Algorithm 1, Algorithm 3 also includes a number of key optimizations detailed below. Observe that for these techniques to be easily integrated, the algorithm *needs* to operate in constructive mode.

A first observation is that the input formula is assumed to be *trimmed*, i.e. the use of iterative identification of unsatisfiable cores was used to reduce the size of the working CNF formula. Clause set trimming is detailed in Section 4.2. To describe the techniques used to improve the performance of MUS extraction, it is convenient to isolate the clauses known to be part of an MUS (i.e. $\mathcal{M}$) from the clauses yet to be analyzed (i.e. $\mathcal{F}'$). Hence, the algorithm can be viewed as constructive. The new techniques are included in lines 7, 10, and 12.

The first technique (line 7) consists of creating a more constrained instance of SAT, by adding to the CNF formula the negation of the removed clause. It is well-known that c_i is redundant if $\mathcal{F} \setminus \{c_i\} \cup \{\neg c_i\}$ is unsatisfiable [19]. Although this technique was first proposed elsewhere [27], in the context of an $\mathcal{O}(m \times k)$ algorithm for MUS extraction, it has not been used in destructive (or hybrid) MUS extraction algorithms. In addition, its use affects the integration of other techniques, as discussed below.

Next, we analyze the technique summarized in line 12 of Algorithm 3. First, assume that the redundancy removal technique is not used, i.e. $\{\neg c_i\}$ is not added to the CNF formula given to the SAT solver. Let the outcome of the SAT solver be false. In this case, one can *refine* the working set of clauses with the unsatisfiable subformula computed by the SAT solver.

Lemma 5 (Clause Set Refinement). *Let $\mathcal{F}$, $\mathcal{F}'$, $\mathcal{M}$ and $\mathcal{U}$ be as defined in Section 2. Consider the outcome of the SAT solver on formula $\mathcal{F}' \cup \mathcal{M}$. If the result is unsatisfiable, with unsatisfiable subformula $\mathcal{U}$, then any MUS in $\mathcal{U}$ contains $\mathcal{M}$. Thus, the working formula $\mathcal{F}'$ can be set to $\mathcal{U} \setminus \mathcal{M}$.*

Algorithm 3: Hybrid MUS Extraction

```
Input  : (Trimmed) Unsatisfiable CNF Formula $\mathcal{F}$
Output: MUS $\mathcal{M}$
begin
    $\mathcal{F}' \leftarrow \mathcal{F}$                                   // Working CNF formula
    $\mathcal{M} \leftarrow \emptyset$                                   // MUS under-approximation
    while $\mathcal{F}' \neq \emptyset$ do
        $c_i \leftarrow$ GetClause($\mathcal{F}'$)
        $\mathcal{F}' \leftarrow \mathcal{F}' \setminus \{c_i\}$
        $(\mathsf{st}, \nu, \mathcal{U}) = \mathtt{SAT}(\mathcal{M} \cup \mathcal{F}' \cup \{\neg c_i\})$      // Add redundancy checking
        if $\mathsf{st} = \mathsf{true}$ then                  // If SAT, $c_i$ is transition clause
            $\mathcal{M} \leftarrow \mathcal{M} \cup \{c_i\}$
            $(\mathcal{F}', \mathcal{M}) \leftarrow$ Rotate($\mathcal{F}', \mathcal{M}, \nu$)   // Find more transition clauses
        else if $\mathcal{U} \subseteq \mathcal{M} \cup \mathcal{F}'$ then      // Equivalently, if $\mathcal{U} \cap \{\neg c_i\} = \emptyset$
            $\mathcal{F}' \leftarrow \mathcal{U} \setminus \mathcal{M}$                       // Clause-set refinement
    return $\mathcal{M}$                                        // Final $\mathcal{M}$ is an MUS
end
```

Proof. By construction, $\mathcal{M}$ is composed of transition clauses, each of which is part of an MUS (see Lemma 1). Hence, any MUS in $\mathcal{U}$ must contain the clauses in $\mathcal{M}$. Since the clauses in $\mathcal{M}$ are known to be transition clauses, the working formula $\mathcal{F}'$ can be updated to $\mathcal{U} \setminus \mathcal{M}$. □

A more complicated version of clause set refinement, that involves considering the resolution proof after each unsatisfiable outcome, has been described elsewhere [6, 25]. Our approach considers solely the computed unsatisfiable core, and so allows using the SAT solver as a black box (provided the solver returns an unsatisfiable core).

The integration of the redundancy removal technique (line 7) and clause set refinement is not immediate. The solution is to provide a test (line 11) to decide when the unsatisfiable core can be used as the next working CNF formula.

Proposition 1. *Let $\mathcal{U}$ be the unsatisfiable core returned by the SAT solver in line 7 of Algorithm 3. If $\mathcal{U} \cap \{\neg c_i\} = \emptyset$, then $\mathcal{U}$ contains an MUS $\mathcal{S}$ of $\mathcal{F}$.*

Finally, we analyze the technique summarized in line 10 of Algorithm 3. Let the outcome of the SAT solver be true and let ν be the computed model. This assignment *must* unsatisfy the clause removed from $\mathcal{F}'$. Similarly, *any* assignment that unsatisfies a *single* clause c_k from $\mathcal{F}'$ and satisfies all clauses in $\mathcal{M}$ *proves* that c_k must be part of an MUS.

Lemma 6. *Let $\mathcal{F}$, $\mathcal{F}' \subseteq \mathcal{F}$ and $\mathcal{M}$ be as defined in Section 2. Let ν be a model of $\mathcal{M} \cup \mathcal{F}' \cup \{\neg c_i\}$ (that* must *unsatisfy clause c_i). Then c_i is included in any MUS of $\mathcal{F}$ that contains $\mathcal{M}$.*

Proof. c_i is a transition clause. Hence, by Lemma 1, c_i is included in *any* MUS of $\mathcal{F}'$. Since $\mathcal{F}' \subseteq \mathcal{F}$, any MUS of $\mathcal{F}'$ is an MUS of $\mathcal{F}$. □

Therefore, given a model ν, we can compute additional clauses to add to the MUS by selective flipping of the variable assignments in ν. The question is then how to decide which variable assignments to flip. The technique described in this paper is referred to as *model rotation*. This technique consists of analyzing changes to the computed model ν that will satisfy the single clause unsatisfied by ν. In order to keep the overhead low, only *single* literal flips are considered. This is illustrated with the following example.

Example 1 (Model Rotation). Let $\mathcal{F} = \{c_1, c_2, c_3, c_4\}$ be an unsatisfiable formula, with $c_1 = \{\neg x_1, x_2\}$, $c_2 = \{\neg x_1, \neg x_2\}$, $c_3 = \{x_1\}$, and $c_4 = \{\neg x_2, x_1, x_3\}$. Also, let $\mathcal{M} = \emptyset$. Suppose that c_1 is removed from $\mathcal{F}$. Then $\mathcal{F} \setminus \{c_1\}$ is satisfiable, with model $\nu = \{x_1, \neg x_2\}$. This means that c_1 is part of an MUS, and so it is added to $\mathcal{M}$. Observe that this model (necessarily) unsatisfies c_1. The next step is to check whether a literal flip in ν unsatisfies *exactly* another clause. For this example, flipping $\neg x_2$ to x_2 satisfies c_1 and solely unsatisfies c_2. This means that c_2 is also part of an MUS of $\mathcal{F}$. The resulting model of $\mathcal{M} \cup \mathcal{F} \setminus \{c_2\}$ is $\nu' = \{x_1, x_2\}$, and $\mathcal{M}$ is updated to $\{c_1, c_2\}$. We can now analyze ν' and check for a single flip that satisfies c_2 and unsatisfies a single clause of the remaining clauses not already in $\mathcal{M}$, namely c_3 and c_4. For example, flipping x_1 to $\neg x_1$ satisfies c_2 and unsatisfies c_3. Since c_3 is the solely unsatisfied clause, then c_3 is also part of an MUS of $\mathcal{F}$. The resulting model of $\mathcal{M} \cup \mathcal{F} \setminus \{c_3\}$ is $\nu'' = \{\neg x_1, x_2\}$, and $\mathcal{M}$ is updated to $\{c_1, c_2, c_3\}$. Observe that the model cannot be further rotated, since $\mathcal{M} = \{c_1, c_2, c_3\}$ is already unsatisfiable. This also means that c_4 is excluded from the computed MUS.

Clearly, model rotation could use more elaborate approaches for finding assignments that unsatisfy a single clause. For example, local search or even a complete SAT solver could be considered. Nevertheless, the objective of model rotation is to eliminate calls to the SAT solver, and so a simple (linear time) procedure is used instead. The analysis of computed models was first used in [27]. However, model rotation is a fundamentally different technique. Whereas the approach in [27] associates a model with each clause and requires worst-case quadratic space, model rotation simply considers single variable value changes to each computed model, so as to identify clauses that are in an MUS of the original formula.

Our results indicate that model rotation is a very effective technique, often allowing a large percentage of the satisfiable SAT calls to be skipped. Clearly, it is far more efficient to evaluate possible model rotations (in linear time), than to modify the SAT instance and call the SAT solver (in worst-case exponential time). This observation holds even if the problem instance is easy to solve.

Although the techniques described in this section are integrated in Algorithm 3, they can be applied with minor modifications to any destructive, constructive or dichotomic MUS algorithm.

4.1 Analysis of Other Techniques

Algorithm 3 integrates, adapts and extends several techniques proposed in earlier work. One additional technique could be considered, namely autarkies [17]. For example, autarkies have been successfully used in recent MUS enumeration algorithms [21]. In contrast, the use of autarkies in Algorithm 3 is less clear. First, by definition a clause is part of an autarky if and only if it is not included in *any* resolution refutation. Hence, since the proposed algorithms start by trimming the initial CNF formula, the autarkies of $\mathcal{F}$ are guaranteed to be *automatically* removed. Nevertheless, a less known observation is that, since clauses are discarded while searching for an MUS, it is possible that additional autarkies may exist with respect to $\mathcal{F}'$. Nevertheless, and similarly to clause set trimming, the use of clause set refinement also *guarantees* that autarkies are automatically eliminated, and so need not be computed. Although the previous observations suggest that identification of autarkies is unnecessary if clause set trimming and refinement are used, there are cases where autarkies *can* still find application in Algorithm 3. Observe that, due to the redundancy removal technique, clause set refinement may not be applicable after every unsatisfiable outcome. When this happens, then autarkies may exist, and can be identified. However, our experimental results indicate that the size of new autarkies does not justify their computation during the execution of the MUS extraction algorithm.

4.2 Preprocessing & Interfacing SAT Solvers

As indicated earlier, a standard technique for computing MUSes of large CNF formulas is *clause set trimming*, that consists of iteratively calling the SAT solver on computed unsatisfiable subformulas until no changes are detected in between calls to the SAT solver [28]. However, for large practical problem instances, iterating the computation of unsatisfiable subformulas until a fixed point is reached can be inefficient. A simpler alternative is to iterate the computation of unsatisfiable subformulas a constant number of times, or until the size change in the computed unsatisfiable subformulas is below a given threshold. Observe that clause set trimming can be viewed as the preprocessing step equivalent to clause set refinement described earlier in Section 4.

In MUS extraction algorithms, SAT solvers can either be used in incremental or non-incremental mode (e.g. [2]). Recent experimental results suggest that incremental mode provides significant performance gains [27, 25]. Our implementation uses an incremental interface to the SAT solver, with one key change. Any clause c_i declared as being part of the MUS $\mathcal{M}$ needs not continue to be handled in incremental mode. Hence, the assumption variable used to activate c_i can be eliminated. This technique is beneficial for problem instances with large MUSes, since the overhead of the incremental interface is reduced as more clauses are added to the MUS $\mathcal{M}$.

5 Results

The algorithms described in the previous sections were implemented in the MUS extraction tool MUSer (MUS ExtratoR), built on top of the Picosat [2] SAT solver. Supported by existing experimental evidence [23], the incremental interface of Picosat was used. (Observe that other work [25] also proposes the use of the incremental interface of modern SAT solvers.) The experimental evaluation focused on the following MUS extractors: the new constructive MUS extraction algorithm based on relaxation variables (*CRV*) described in section 3; the hybrid MUS extraction algorithm (*HYB*) described in section 4; a reference destructive algorithm (*DREF*); a reference constructive algorithm [5] (*CREF*); the recent constructive algorithm from [27] (*MUNSAT*); a recent local-search-guided destructive MUS extraction algorithm from [11] (*AOMUS*);

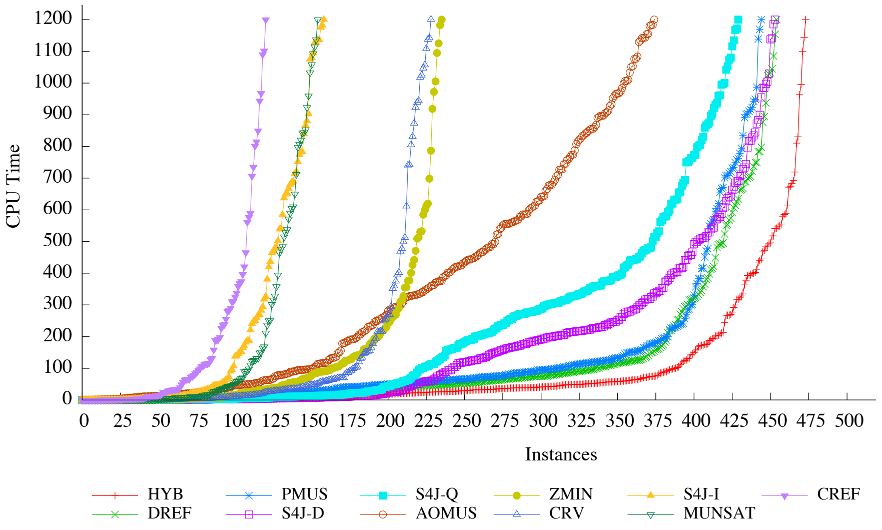

Fig. 1. Cactus plot with running times of MUS extractors

a well-known MUS extractor from [28] (ZMIN); SAT4J [18] MUS extractor in linear constructive mode (*S4J_I*), in QuickXPlain [16] mode (*S4J_Q*), and in destructive mode (*S4J_D*). Finally, a destructive MUS extraction algorithm available in the Picosat distribution [2] (PMUS). As shown by the results below, fairly recent MUS extractors [11, 27, 7] perform considerably worse than the most recent generation of MUS extractors, including the ones described in this paper.

The experimental evaluation focused on 500 problem instances submitted to the upcoming MUS track of the 2011 SAT Competition [3]. All problem instances were obtained from practical applications of SAT, including hardware bounded model checking, FPGA routing, hardware & software verification, equivalence checking, abstraction refinement, design debugging, function decomposition, and bioinformatics. Clause set trimming was applied to all problem instances before running *any* of the MUS extraction algorithms. Otherwise, algorithms that do not implement clause set trimming would perform poorly. All results were obtained on an HPC cluster, where each node is an 8-core CPU Xeon E5450 3GHz, with 32GByte RAM and running Linux. For each problem instance, the specified resources were a time limit of 1200 seconds and a memory limit of 4 GByte. For SAT4J, the Java virtual machine used was the Java HotSpot(TM) 64-Bit Server VM (build 19.1-b02). Figure 1 shows a cactus plot with all MUS extractors, showing the instances solved by increasing run times. The following conclusions can be drawn. First, the new constructive algorithm based on relaxation variables (CRV) clearly outperforms all other constructive algorithms, namely MUNSAT, S4J_C and CREF. Second, and more importantly, the new hybrid algorithm *HYB* outperforms all other MUS extraction algorithms. It solves more instances, but the plot also shows a clear performance edge with respect to all other algorithms. Third, fairly recent MUS extractors algorithms, namely MUNSAT [27] and AOMUS [11], perform

[3] http://www.satcompetition.org/2011/.

Table 1. Number of solved instances

Solver	CREF	MUNSAT	S4J_I	CRV	ZMIN	AOMUS	S4J_Q	PMUS	S4J_D	DREF	HYB
# Solved	112	154	158	228	235	374	429	444	453	454	**473**

Table 2. Comparison with [25]

Instance	3pipe	4pipe_1	barrel6	barrel7	barrel8	longmult6	longmult7	longmult8
Best in [25]	**167**	1528	348	700	4110	968	5099	—
HYB	194	**1143**	**35**	**72**	400	**11**	**99**	**811**
DREF	365	—	40	94	**332**	30	398	—
PMUS	—	—	68	102	701	51	283	—
S4J_S	223	—	395	829	—	152	883	—

significantly worse than the more recent generation of MUS extractors. Fourth, and finally, constructive algorithms perform significantly worse than destructive algorithms, the exceptions being the new algorithms *CRV* and *HYB*. However, the results confirm that constructive algorithms requiring $\mathcal{O}(m \times k)$ calls to a SAT solver simply do not scale in practice.

The cactus plot is completed with Table 1, that shows the number of solved instances. The main conclusions here are that: (i) the new algorithm *HYB* solves the largest number of instances; and (ii) recently published MUS extraction algorithms [11, 27] are unable to solve many instances, many of which are easily solved by other approaches.

Finally, Figure 2 shows scatter plots comparing the run times of *HYB* with the next best MUS extraction algorithms, namely *DREF*, *S4J_D*, *PMUS*, and *AOMUS*. Again the results are clear. *HYB* clearly outperforms *DREF*, i.e. the reference implementation of destructive MUS extraction. Moreover, *HYB* clearly outperforms *PMUS*, in many cases by one order of magnitude or more. Also, *HYB* extensively outperforms *AOMUS*, in most cases by more than one order of magnitude. Finally, *HYB* also outperforms *S4J_D*, although in this case there are a number of outliers. These outliers represent problem instances with small MUSes, for which *S4J_D* performs well.

To conclude the experimental evaluation, the best performing MUS extraction tools are compared against the MUS extractor from [25], on selected problem instances. The best run times from [25] are used, since the tool is not publicly available. Moreover, the hardware where the MUS extractors were run is similar. The run times (in seconds) are shown in Table 2. As can be concluded, *HYB* performs significantly better. For the *barrel* instances, the speedup is around one order of magnitude. For the *longmult* instances, the speedup is almost two orders of magnitude. For the *pipe* instances, *HYB* performs better in one instance, and worse in another.

6 Related Work

To the best of our knowledge, Algorithm 2 is new. Nevertheless, the use of relaxation variables for MUS extraction has been proposed in earlier work. For example, AMUSE [26] also uses relaxation variables. However, AMUSE does not compute an

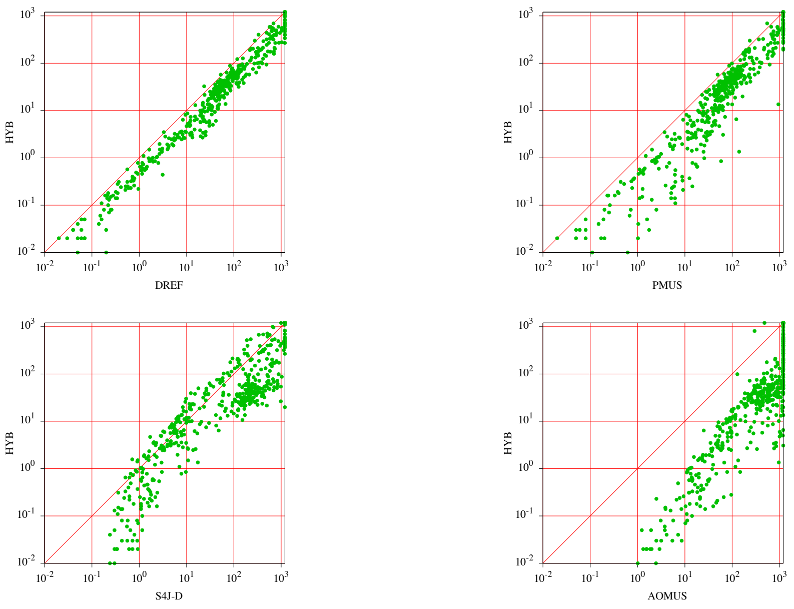

Fig. 2. Scatter plot comparing HYB with other MUS extractors

MUS, and identifies instead a reduced unsatisfiable subset. The use of relaxation variables has also been considered extensively in the enumeration of MUSes [20, 22], and in the use of MaxSAT for MUS extraction [7]. Although the use of relaxation variables resembles the use of selector variables [25], it is *fundamentally* different. Selector variables serve *solely* to specify clause (de)activation in incremental SAT. Relaxation variables serve to specify constraints on how many clauses can be relaxed.

Algorithm 3 is novel, even though its organization can be viewed as a (constructive) variant of Algorithm 1. Moreover, some of the techniques implemented by Algorithm 3 are novel, and their integration is also novel. Also, the implementation of these techniques requires a constructive MUS extraction algorithm. Clause set refinement was first studied in [6, 25]. However, the solution proposed there is more complicated, being based on analyzing resolution proofs. In contrast, our approach simply uses the returned unsatisfiable core. The analysis of computed models for finding more than one transition clause per iteration of the algorithm was first used in [27], in the context of a constructive algorithm requiring $\Theta(m \times k)$ calls to a SAT solver. In [27], each clause is characterized by an *associated assignment*, that aims to satisfy all clauses in a working set of clauses but itself; clearly this can entail non-negligible memory requirements for large-scale problems instances. The model rotation technique proposed in this paper is novel, since computed models are only analyzed immediately after being computed, and only checked for single changes of variable values. Finally, the technique of including $\{\neg c_i\}$ in the CNF formula given to the SAT solver is standard in CNF redundancy checking [19], and was first used for MUS extraction in [27]. Our implementation follows this approach. Nevertheless, this paper proposes a new solution for integrating the redundancy removal technique and clause set refinement.

7 Conclusions

This paper develops new algorithms for the efficient extraction of MUSes from unsatisfiable CNF formulas, and has two main contributions. The first contribution is a new constructive MUS extraction algorithm. Whereas existing algorithms require $\mathcal{O}(m \times k)$ calls to a SAT oracle, the new algorithm requires $\mathcal{O}(m)$ calls. In practice, the new algorithm is shown to outperform all existing constructive algorithms. More importantly, this new algorithm shows that constructive and destructive MUS extraction algorithms share a number of important similarities. The second contribution exploits this observation, and develops a hybrid algorithm, that is organized as a constructive algorithm, but that exploits features of destructive algorithms. In addition, this algorithm integrates a number of key MUS extraction techniques, including redundancy removal, clause set refinement, and model rotation, that essentially exploit *all* of the main steps of the MUS extraction algorithm, i.e. calls to the SAT solver, and both unsatisfiable and satisfiable outcomes. Moreover, the paper also develops conditions for the integration of these techniques. Although these techniques are integrated in the new algorithm, they can be used with any MUS extraction algorithm. The resulting algorithm (*HYB*) outperforms all publicly available MUS extraction tools. The performance gains often exceed one order of magnitude when compared with state of the art MUS extraction tools. Finally, algorithm *HYB* is shown to also outperform recent non-publicly available MUS extraction algorithms [25].

The experimental results are promising and indicate that *HYB* represents the new state of the art in the area of MUS extraction algorithms. Nevertheless, practical applications of MUS extraction algorithms can gain from more efficient solutions. Envisioned research directions include better heuristics for model rotation and adapting SAT solvers to minimize computed unsatisfiable subformulas, e.g. by exploiting the AMUSE [26] heuristics.

Acknowledgement. This work is partially supported by SFI PI grant BEACON (09/IN.1/I2618), European Community FP7 project MANCOOSI (214898) and FCT through grants ATTEST (CMU-PT/ELE/0009/2009) and ASPEN (PTDC/EIA-CCO/110921/2009), and INESC-ID multiannual funding from the PIDDAC program funds.

References

1. R. R. Bakker, F. Dikker, F. Tempelman, and P. M. Wognum. Diagnosing and solving over-determined constraint satisfaction problems. In *International Joint Conference on Artificial Intelligence*, pages 276–281, 1993.
2. A. Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:75–97, 2008.
3. R. Bruni. On exact selection of minimally unsatisfiable subformulae. *Ann. Math. Artif. Intell.*, 43(1):35–50, 2005.
4. J. W. Chinneck and E. W. Dravnieks. Locating minimal infeasible constraint sets in linear programs. *INFORMS Journal on Computing*, 3(2):157–168, 1991.
5. J. L. de Siqueira N. and J.-F. Puget. Explanation-based generalisation of failures. In *European Conference on Artificial Intelligence*, pages 339–344, 1988.

6. N. Dershowitz, Z. Hanna, and A. Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *Theory and Applications of Satisfiability Testing*, pages 36–41, 2006.
7. C. Desrosiers, P. Galinier, A. Hertz, and S. Paroz. Using heuristics to find minimal unsatisfiable subformulas in satisfiability problems. *J. Comb. Optim.*, 18(2):124–150, 2009.
8. C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. Satisfiability solvers. In *Handbook of Knowledge Representation*, pages 89–134. Elsevier, 2008.
9. É. Grégoire, B. Mazure, and C. Piette. Extracting MUSes. In *European Conference on Artificial Intelligence*, pages 387–391, August 2006.
10. É. Grégoire, B. Mazure, and C. Piette. Boosting a complete technique to find MSS and MUS thanks to a local search oracle. In *International Joint Conference on Artificial Intelligence*, pages 2300–2305, January 2007.
11. É. Grégoire, B. Mazure, and C. Piette. Local-search extraction of MUSes. *Constraints*, 12(3):325–344, 2007.
12. É. Grégoire, B. Mazure, and C. Piette. On approaches to explaining infeasibility of sets of Boolean clauses. In *International Conference on Tools with Artificial Intelligence*, pages 74–83, November 2008.
13. É. Grégoire, B. Mazure, and C. Piette. Using local search to find MSSes and MUSes. *European Journal of Operational Research*, 199(3):640–646, 2009.
14. F. Hemery, C. Lecoutre, L. Sais, and F. Boussemart. Extracting MUCs from constraint networks. In *European Conference on Artificial Intelligence*, pages 113–117, 2006.
15. J. Huang. MUP: a minimal unsatisfiability prover. In *Asia South Pacific Design Automation*, pages 432–437, 2005.
16. U. Junker. QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In *AAAI Conference on Artificial Intelligence*, pages 167–172, 2004.
17. O. Kullmann. Lean clause-sets: generalizations of minimally unsatisfiable clause-sets. *Discrete Applied Mathematics*, 130(2):209–249, 2003.
18. D. Le Berre and A. Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.
19. P. Liberatore. Redundancy in logic I: CNF propositional formulae. *Artif. Intell.*, 163(2):203–232, 2005.
20. M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008.
21. M. H. Liffiton and K. A. Sakallah. Searching for autarkies to trim unsatisfiable clause sets. In *Theory and Applications of Satisfiability Testing*, pages 182–195, 2008.
22. M. H. Liffiton and K. A. Sakallah. Generalizing core-guided Max-SAT. In *Theory and Applications of Satisfiability Testing*, pages 481–494, 2009.
23. J. Marques-Silva. Minimal unsatisfiability: Models, algorithms and applications. In *International Symposium on Multiple-Valued Logic*, pages 9–14, 2010.
24. J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *SAT Handbook*, pages 131–154. IOS Press, 2009.
25. A. Nadel. Boosting minimal unsatisfiable core extraction. In *Formal Methods in Computer-Aided Design*, October 2010.
26. Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. AMUSE: a minimally-unsatisfiable subformula extractor. In *Design Automation Conference*, pages 518–523, 2004.
27. H. van Maaren and S. Wieringa. Finding guaranteed MUSes fast. In *Theory and Applications of Satisfiability Testing*, pages 291–304, 2008.
28. L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe Conference*, pages 10880–10885, 2003.