Using Minimal Correction Sets to More Efficiently Compute Minimal Unsatisfiable Sets

Fahiem Bacchus[1](✉) and George Katsirelos[2]

[1] Department of Computer Science, University of Toronto, Toronto, ON, Canada
fbacchus@cs.toronto.edu
[2] MIAT, INRA, Toulouse, France
george.katsirelos@toulouse.inra.fr

Abstract. An unsatisfiable set is a set of formulas whose conjunction is unsatisfiable. Every unsatisfiable set can be corrected, i.e., made satisfiable, by removing a subset of its members. The subset whose removal yields satisfiability is called a correction subset. Given an unsatisfiable set $\mathcal{F}$ there is a well known hitting set duality between the unsatisfiable subsets of $\mathcal{F}$ and the correction subsets of $\mathcal{F}$: every unsatisfiable subset hits (has a non-empty intersection with) every correction subset, and, dually, every correction subset hits every unsatisfiable subset. An important problem with many applications in practice is to find a minimal unsatisfiable subset (MUS) of $\mathcal{F}$, i.e., an unsatisfiable subset all of whose proper subsets are satisfiable. A number of algorithms for this important problem have been proposed. In this paper we present new algorithms for finding a single MUS and for finding all MUSES. Our algorithms exploit in a new way the duality between correction subsets and unsatisfiable subsets. We show that our algorithms advance the state of the art, enabling more effective computation of MUSES.

1 Introduction

A set of formulas is said to be unsatisfiable if the conjunction of its members has no model (is unsatisfiable). A minimal unsatisfiable set (a MUS) has the additional property that every proper subset of it is satisfiable.

Given an unsatisfiable set $\mathcal{F}$ the task of computing a MUS contained in $\mathcal{F}$ (a MUS of $\mathcal{F}$) has long been an important problem for a range of verification applications related to diagnosis and debugging, e.g., program type debugging, circuit diagnosis, production configuration (see [6]).

MUSES have become even more important with the increasing applications of SAT based approaches in system analysis and verification. In [23] a number of ways that MUSES can be used in SAT based bounded model checking (BMC) are presented. For example, a MUS might tell the user that the property being checked did not play a role in deriving *unsat*, thus indicating that the system specification is unconstrained. MUSES also play an important role in applications that exploit unsatisfiable sets (sometimes called unsatisfiable cores).

D. Kroening and C.S. Păsăreanu (Eds.): CAV 2015, Part II, LNCS 9207, pp. 70–86, 2015.
DOI: 10.1007/978-3-319-21668-3_5

As discussed in [6] many of these application can benefit significantly from computing MUSES rather than just using the default unsatisfiable core returned by the solver. Formal equivalence checking, proof-based abstraction refinement, and boolean function bi-decomposition are three important applications in which computing a MUS has proved to be beneficial [6]. Belov et al. [4] present some more recent results quantifying the benefits of computing MUSES in the hybrid counterexample/proof-based abstraction engine GLA implemented in the ABC verification tool [9]. A more recent application of MUSES arises in the Franken-Bit verifier [12] where MUSES are used to compute invariants [13].

With this range of applications it is not surprising that there has been an extensive amount of research into developing more effective algorithms for computing MUSES, e.g., [5,6,11,14,18–20] (see [6] for a more extensive list).

In this paper we continue this line of research and present new algorithms for computing MUSES. Our new algorithms exploit the well known hitting set duality between the unsatisfiable subsets of an unsatisfiable set $\mathcal{F}$ and the correction subsets of $\mathcal{F}$. Our algorithms work in particular with *minimal subsets*—the duality also holds between minimal unsatisfiable subsets and minimal correction subsets. This duality has been exploited before to compute all MUSES in the CAMUS system [16]. However, in CAMUS the first step was to compute the set of all MCSES, *AllMcses*, from which all MUSES can be extracted by finding all minimal hitting sets of *AllMcses*. Unfortunately in practice it is often impossible to complete the first step of computing *AllMcses*.

We find new ways to exploit the MUS/MCS connection in order to compute a single MUS and to incrementally compute all MUSES. Our method does not require computing *AllMcses*. We show empirically that our new algorithms advance the state of the art in MUS computation, and thus can potentially enhance a range of applications in formal methods that rely on computing MUSES.

2 Background

Let $\mathbb{T}$ be some background theory and $\mathcal{F}$ be a set of $\mathbb{T}$-formulas such that the **conjunction** of these formulas is $\mathbb{T}$-*unsat*, i.e., their conjunction has no $\mathbb{T}$-*model*. In many applications it is useful to identify a smaller subset of $\mathcal{F}$ that is $\mathbb{T}$-*unsat*. In practice, if the $\mathbb{T}$-*sat* status of various subsets of $\mathcal{F}$ can be effectively determined, then finding a **minimal** subset of $\mathcal{F}$ that is $\mathbb{T}$-*unsat* is often achievable.

In this paper we will always regard a set of formulas $\mathcal{F}$ as representing the conjunction of its members. So, e.g., $\mathcal{F}$ is $\mathbb{T}$-*unsat* means $\bigwedge_{f\in\mathcal{F}} f$ is $\mathbb{T}$-*unsat*.

Definition 1 (MUS). An unsatisfiable subset U of $\mathcal{F}$ is a subset of $\mathcal{F}$ that is $\mathbb{T}$-*unsat*. A **Minimal Unsatisfiable Set** (MUS) of $\mathcal{F}$ is a unsatisfiable subset $M \subseteq \mathcal{F}$ that is minimal w.r.t. set inclusion. That is, M is $\mathbb{T}$-*unsat* and every proper subset $S \subsetneq M$, S is $\mathbb{T}$-*sat*.

Definition 2 (MSS). A satisfiable subset of $\mathcal{F}$ is a subset of $\mathcal{F}$ that is $\mathbb{T}$-*sat*. A **Maximal Satisfiable Subset** (MSS) of $\mathcal{F}$ is a satisfiable subset $S \subseteq \mathcal{F}$ that is maximal w.r.t set inclusion. That is, S is $\mathbb{T}$-*sat* and for every proper superset $U \supsetneq S$ such that $U \subseteq \mathcal{F}$, U is $\mathbb{T}$-*unsat*.

Definition 3 (MCS). A correction subset of $\mathcal{F}$ is a subset of $\mathcal{F}$ whose complement in $\mathcal{F}$ is $\mathbb{T}$-*sat*. A **Minimal Correction Subset** (MCS) of $\mathcal{F}$ is a correction subset $C \subseteq \mathcal{F}$ that is minimal w.r.t. set inclusion, i.e., $\mathcal{F} \setminus C$ is an MSS of $\mathcal{F}$.

Definition 4. A formula $f \in \mathcal{F}$ is said to be **critical** (or a *transition* formula [7]) for $\mathcal{F}$ when $\mathcal{F}$ is $\mathbb{T}$-*unsat* and $\mathcal{F} - \{f\}$ is $\mathbb{T}$-*sat*.

Intuitively, a MUS is an unsatisfiable set that cannot be reduced without causing it to become satisfiable; a MSS is a satisfiable set that cannot be added to without causing it to become unsatisfiable; and an MCS is a minimal set of removals from $\mathcal{F}$ that causes $\mathcal{F}$ to become satisfiable.

A critical formula for $\mathcal{F}$ is one whose removal from $\mathcal{F}$ causes $\mathcal{F}$ to become satisfiable. It should be noted if f is critical for $\mathcal{F}$ then (a) f must be contained in every MUS of $\mathcal{F}$ and (b) $\{f\}$ is an MCS of $\mathcal{F}$. Furthermore, it can be observed that M is a MUS if and only if every $f \in M$ is critical for M. Note that a formula f that is critical for a set S is not necessarily critical for a superset $S' \supset S$. In particular, S' might contain other MUSES that do not contain f.

Duality. There is a well known hitting set duality between MUSES and MCSES that to the best of our knowledge was first presented formally by Reiter [22] in the context of diagnosis problems.

A hitting set H of a collection of sets $\mathcal{C}$ is a set that "hits" every set in $\mathcal{C}$ in the sense that it has a non empty intersection with each such set: $\forall C \in \mathcal{C}.H \cap C \neq \emptyset$. A hitting set H is minimal (or irreducible) if no subset of H is a hitting set.

Let *AllMuses*$(\mathcal{F})$ (*AllMcses*$(\mathcal{F})$) be the set containing all MUSES (MCSES) of $\mathcal{F}$. Then Reiter's result can be recast to show that $M \in$ *AllMuses*$(\mathcal{F})$ iff M is a minimal hitting set of *AllMcses*$(\mathcal{F})$, and dually, $\mathcal{C} \in$ *AllMcses*$(\mathcal{F})$ iff C is a minimal hitting set of *AllMuses*$(\mathcal{F})$. Intuitively, we can see that if a MUS M fails to hit an MCS C, then $M \subseteq \mathcal{F}{-}C$, i.e., M is a subset of a satisfiable set and hence can't be unsatisfiable. Similarly, if an MCS C fails to hit a MUS M then $\mathcal{F}{-}C \supseteq M$ is a superset of an unsatisfiable set and hence can't be satisfiable. It is also not hard to see that the duality between MCSES and MUSES also holds for non-minimal sets. That is, every correction subset (not necessarily minimal) hits all unsatisfiable subsets and vice versa.

Although we have discussed MUSES and MCSES in the context of a fixed set of formulas $\mathcal{F}$ we will also be working with subsets of $\mathcal{F}$. It is useful to point out that if $\mathcal{F}' \subseteq \mathcal{F}$, then *AllMuses*$(\mathcal{F}') \subseteq$ *AllMuses*$(\mathcal{F})$ and in general *AllMuses*$(\mathcal{F}') \neq$ *AllMuses*$(\mathcal{F})$. Hence, if f is critical for $\mathcal{F}$ it is critical for all unsatisfiable subsets of $\mathcal{F}$ (f is critical iff it is contained in every MUS).

An MCS C' of $\mathcal{F}' \subset \mathcal{F}$ is not necessarily an MCS of $\mathcal{F}$, however C' can always be extended to an MCS C of $\mathcal{F}$. In particular, we can add the formulas of $\mathcal{F} \setminus \mathcal{F}'$ to $\mathcal{F}'$ one at a time. If C' is no longer a correction subset of $\mathcal{F}' \cup \{f\}$ we add f to C'. At each stage the augmented C' is an MCS of the augmented $\mathcal{F}'$, and at the end C' has been extended to be an MCS of $\mathcal{F}$. Since we have not seen this observation previously in the literature, and its proof is illustrative of concepts needed in our algorithms, we provide a proof here.

Proposition 1. *Let $C' \in \mathit{AllMcses}(\mathcal{F}')$ and $f \in \mathcal{F} \setminus \mathcal{F}'$. If C' is a correction subset of $\mathcal{F}' \cup \{f\}$ it is an* MCS *of $\mathcal{F}' \cup \{f\}$, and if it is not then $C' \cup \{f\}$ is an* MCS *of $\mathcal{F}' \cup \{f\}$.*

Proof. C' is a minimal correction subset of $\mathcal{F}'$ if and only if for every $a \in C'$ there exists a MUS $M \in \mathit{AllMuses}(\mathcal{F}')$ such that $M \cap C' = \{a\}$. That is, M is only hit by a, hence C' will no longer be a correction subset if we remove a. M serves as a witness that a is needed in C', and C' is minimal iff every member of C' has a witness. Since $\mathit{AllMuses}(\mathcal{F}') \subseteq \mathit{AllMuses}(\mathcal{F}' \cup \{f\})$, the witnesses for C' remain valid after adding f to $\mathcal{F}'$ and if C' corrects $\mathcal{F}' \cup \{f\}$ it must be an MCS of $\mathcal{F}' \cup \{f\}$. If C' does not correct $\mathcal{F}' \cup \{f\}$ then there are some MUSES in $\mathit{AllMuses}(\mathcal{F}' \cup \{f\})$ that are not hit by C'. But since C' hits all muses in $\mathit{AllMuses}(\mathcal{F}')$ these un-hit MUSES must contain f. So $C' \cup \{f\}$ is a correction subset of $\mathcal{F}' \cup \{f\}$. Furthermore, any of these new MUSES can serve as a witness for f, and for every $a \in C$ there is a witness for a in $\mathit{AllMuses}(\mathcal{F}')$ which cannot contain f. Hence, these witnesses remain valid when f is added to C', and $C' \cup \{f\}$ is an MCS of $\mathcal{F}' \cup \{f\}$. $\square$

Although we have given the above definitions in terms of an arbitrary theory $\mathbb{T}$, in the rest of this paper we will work with $\mathbb{T}$ being ordinary propositional logic (*Prop*) and $\mathcal{F}$ being a set of clauses. In particular, our algorithms assume access to some basic facilities of modern SAT solvers. Some of these facilities are also available in modern SMT solvers, and thus some of our ideas could be lifted to theories handled by SMT solvers.

3 Prior MUS Algorithms

Current state-of-the-art algorithms for computing MUSES have converged on versions of Algorithm 1.

Algorithm 1 operates on a working set of clauses $W = (\mathit{unkn} \cup \mathit{crits})$ with the clauses of unknown status, *unkn*, initially equal to $\mathcal{F}$. In the main **while** loop the status of each clause in *unkn* is resolved and its size reduced until $\mathit{unkn} = \emptyset$. At this point W consists only of a set of clauses, *crits*, all of which are known to be critical for W. As observed above this implies that $W = \mathit{crits}$ is a MUS.

The input assumption is that $W = \mathcal{F}$ is *unsat*, and this condition is an invariant of the algorithm. Each iteration of the main loop selects a clause of unknown status $c \in \mathit{unkn}$ and tests the satisfiability of $W \setminus \{c\}$. We have that $W \setminus \{c\} \models \neg c$, as W has no models. Hence, we can make the SAT test of $W \setminus \{c\}$ more efficient by adding the implied $\neg c$ (since c is a clause, $\neg c$ is a set of unit clauses which are particularly advantageous for a SAT solver).

If $W \setminus \{c\}$ is *sat* then we know that c is critical for W (and for all subsets of W that the algorithm subsequently works with). In this case we can additionally find more critical clauses by applying the technique of recursive model rotation (RMR) [7,24]. Note that the satisfying model π returned is such that $\pi \models (\mathit{crits} \cup \mathit{unkn}) \setminus \{c\}$ and $\pi \not\models c$, which is the condition required for RMR to work correctly.

Algorithm 1. findmus($\mathcal{F}$): Current state-of-the-art algorithm for computing a MUS

```
Input: $\mathcal{F}$ an unsatisfiable set of clauses
Output: a MUS of $\mathcal{F}$
1  $crits \leftarrow \emptyset$
2  $unkn \leftarrow \mathcal{F}$
3  while $unkn \neq \emptyset$ do
4      $c \leftarrow$ choose $c \in unkn$
5      $unkn \leftarrow unkn \setminus \{c\}$
6      (sat?,$\pi$,$\kappa$) $\leftarrow$ SatSolve($crits \cup unkn \cup \{\neg c\}$)
       /* SatSolve returns the status (sat or unsat), a model $\pi$ if sat, or an unsat
          subset $\kappa$ of the input if unsat.                                        */
7      if sat? then
8          $crits \leftarrow crits \cup \{c\}$
9          $C \leftarrow$ ERMR ($c$, $crits$, $unkn$, $\pi$)
10         $crits \leftarrow crits \cup C$
11         $unkn \leftarrow unkn \setminus C$
12     else
13         if $\kappa \subseteq (crits \cup unkn)$ then
14             $unkn \leftarrow unkn \cap \kappa$
15 return $crits$
```

Every newly identified critical clause is removed from *unkn* and added to *crits* thus reducing the number of iterations of the main loop.

If $W \setminus \{c\}$ is *unsat* then there is some MUS of W that does not contain c. The algorithm then focuses on finding one of these MUSES by removing c from W. Note that there might also be MUSES of W that do contain c so the final MUS found depends on the order in which clauses of *unkn* are tested. One final optimization is that we can obtain an *unsat* core, κ, from the *sat* solver. If that core did not depend on the added $\neg c$ clauses then we can reduce W by setting it to κ. In this case it must be that $crits \subseteq \kappa$: all the clauses of *crits* are critical for $W \setminus \{c\}$. Hence, to make $W = \kappa$ we simply need to remove from *unkn* all clauses not in κ. This optimization is called clause set refinement [20].

Algorithm 1 is used in state of the art MUS finding algorithms like [8,20], and these systems also add a number of other lower level optimizations as described in [20]. The main difference between these MUS finding systems is that some use a modified SAT solver that keeps track of the resolution proof used to derive *unsat* [20]—the unsatisfiable subset κ is extracted from that proof—while others use selector variables for the input clauses and the technique of SAT solving under assumptions to obtain κ [10].

4 MCS Based MUS Finding

In this section we present our new algorithms for finding MUSES. Our algorithms are based on the duality between MCSES and MUSES mentioned in Sect. 2. This duality has been exploited in previous work, in particular in the CAMUS system [16]. However, in that prior work the first step was to compute all MCSES of the input formula $\mathcal{F}$, $\mathit{AllMcses}(\mathcal{F})$, after which MUSES were found by finding minimal hitting sets of $\mathit{AllMcses}(\mathcal{F})$. This first step is very expensive, and sometimes cannot be completed since there can be exponential number of MCSES. So CAMUS is not very effective for the task of finding a single MUS. In this work we revisit this duality to arrive at algorithms that do not require an exhaustive enumeration of all MCSES.

4.1 Finding a Single MUS

Algorithm 2 is our new algorithm for finding a single MUS. Like Algorithm 1, it operates on the working set of clauses $W = (\mathit{unkn} \cup \mathit{crits})$ with the clauses of unknown status, *unkn*, initially equal to $\mathcal{F}$. In the main **while** loop a minimal correction subset of W is computed using Algorithm 3. Algorithm 3 works to find not just any MCS: it searches for an MCS contained entirely in *unkn*. Every clause in the set *crits* is critical for W, and thus every clause in *crits* is a singleton MCS. We are not interested in finding these MCSES. If there is no MCS in *unkn* it must be the case that W remains *unsat* even if all of *unkn* is removed from it. That is, *crits* is an unsatisfiable set all of whose members are critical—it is a MUS.

If we do find an MCS, *CS*, we then choose some clause from it, c, add c to *crits* and remove all of *CS* from *unkn*. Algorithm 3 also returns the satisfying solution, π it found for $W \setminus \mathit{CS}$ (verifying that *CS* is a correction subset). This solution can be used to find more criticals using RMR. Note that since *CS* is a minimal correction subset it must be the case that $\pi \not\models a$ for every $a \in \mathit{CS}$. Thus, $\pi \models (\mathit{crits} \cup \mathit{unkn}) \setminus \{c\}$ and $\pi \not\models c$, which is the condition required for RMR to work correctly. As will be described below we have developed an extension of standard RMR, **em-rmr**, that can find even more new criticals.

Clause set refinement can be used within this algorithm. Algorithm 3 (**find-mcs**) computes an unsatisfiable core whenever $|\mathit{CS}| \geq 1$. From this core an unsatisfiable set $\kappa \subseteq \mathit{crits} \cup \mathit{unkn}$ can be extracted and used as in Algorithm 1 to reduce *unkn* to $\mathit{unkn} \cap \kappa$. A simpler solution, however, is to do another SAT call on the unsatisfiable set $\mathit{crits} \cup \mathit{unkn}$ whenever $|\mathit{CS}| > 1$. In this case the SAT solver has just refuted a closely related formula in **find-mcs** and can exploit its previously learned clauses to quickly refute $\mathit{crits} \cup \mathit{unkn}$. The core it returns can then be intersected with *unkn*. In our experiments, we confirmed that in the vast majority of cases the cost of this step is negligible typically taking less than a second cumulatively.

However, in those cases where the instance contains only one MUS all MCSES will have size 1, and we would never get to perform clause set refinement. We address this deficiency by forcing a SAT call on $\mathit{crits} \cup \mathit{unkn}$ whenever clause set refinement has not been performed for some time. The logic of when to do the SAT call and returning a reduced *unkn* set is encapsulated in the **refine-clause-set** subroutine.

Theorem 1. *If its input formula $\mathcal{F}$ is unsat, **find-mcs** correctly returns an* MCS *of $\mathit{crits} \cup \mathit{unkn}$ contained in unkn if any exist, **em-rmr** correctly returns a set of clauses critical for $\mathit{crits} \cup \mathit{unkn}$, and **refine-clause-set** correctly returns*

Algorithm 2. MCS-MUS($\mathcal{F}$): Find a MUS of $\mathcal{F}$ using MCS duality.

```
Input: $\mathcal{F}$ an unsatisfiable set of clauses
Output: a MUS of $\mathcal{F}$
1  $crits \leftarrow \emptyset$
2  $unkn \leftarrow \mathcal{F}$
3  while $true$ do
4      $(CS, \pi) \leftarrow$ find-mcs$(crits, unkn)$      // Find $CS$, an MCS contained in $unkn$.
5      if $CS = null$ then
6          return $crits$
7      $c \leftarrow$ choose $c \in CS$
8      $crits \leftarrow crits \cup \{c\}$
9      $unkn \leftarrow unkn \setminus CS$
10     $C \leftarrow$ em-rmr $(c, crits, unkn, \pi)$
11     $crits \leftarrow crits \cup C$
12     $unkn \leftarrow unkn \setminus C$
13     $unkn \leftarrow$ refine-clause-set$(crits, unkn, |CS| > 1)$
```

an unsatisfiable subset of $crits \cup unkn$, then Algorithm 2 will return a MUS *of its input formula $\mathcal{F}$.*

Proof. We show that two invariants hold in the main loop of Algorithm 2: (1) $crits \cup unkn$ is unsat and (2) every clause in $crits$ is critical for $crits \cup unkn$.

Algorithm 2 terminates when **find-mcs** is unable to find a correction subset in $unkn$. This happens when $crits \cup unkn$ remains *unsat* even after all the clauses of $unkn$ are removed, i.e., when it detects that $crits$ is *unsat* (see Algorithm 3). In this case, we know that $crits$ is an *unsat* set of clauses and from invariant (2) all of its members are critical, i.e., it is a MUS. Hence, the correctness of Algorithm 2 follows from the invariants.

Initially $crits = \emptyset$ and $unkn = \mathcal{F}$, and $\mathcal{F}$ is *unsat* by assumption. So the invariants hold at the start of the main loop. Assume that they hold up until the $i-1$'th iteration of the main loop. If in the i'th iteration we fail to find an MCS contained in $unkn$, then $crits$ is *unsat* and unchanged from the $i-1$'th iteration. So invariant (1) holds and by induction so does invariant (2).

Otherwise, let CS be the MCS returned by **find-mcs** with $CS \subseteq unkn$. CS is an MCS of $W = crits \cup unkn$, therefore there is a witness $M \in AllMuses(W)$ for every $c \in CS$ with $M \cap CS = \{c\}$. Algorithm 2 updates $crits$ to $crits \cup \{c\}$ (for some $c \in CS$) and $unkn$ to $unkn \setminus CS$. Let this updated set $crits \cup unkn$ be $W' = W \setminus CS \cup \{c\}$. We have that $M \subseteq W'$ so invariant (1) continues to hold. Furthermore, let $M' \in AllMuses(W')$ be any MUS of W'. Since $AllMuses(W') \subseteq AllMuses(W)$, M' is also a MUS of W. Hence M' must be hit by the MCS CS and since W' only contains c from CS we must have $c \in M'$. This shows that c hits all MUSES of W', i.e., removing it from W' removes all MUSES from W' making W' *sat*. That is, c is critical for $W' = crits \cup unkn$, and invariant (2) continues to hold.

Finally since we are assuming that **em-rmr** is correct, the invariants are preserved after **em-rmr** moves some clauses from $unkn$ to $crits$. The call to **refine-clause-set** cannot affect invariant (2) and since we assume that it is correct, it preserves also invariant (1). ■

Algorithm 3. **findmcs**(*crits*, *unkn*): Find an MCS of $crits \cup unkn$ entirely contained in *unkn*.

```
Input: (crits, unkn) Two sets of clauses whose union is unsatisfiable.
Output: CS an MCS of $crits \cup unkn$ that is contained in unkn and a model $\pi$
        such that $\pi \models (crits \cup unkn) \setminus CS$
1  (sat?, $\pi$, $\kappa$) ← SatSolve(crits)
2  if not sat? then
3      return null
4  $CS \leftarrow \{c \in unkn \mid \pi \not\models c\}$
5  while $|CS| > 1$ do
6      (sat?, $\pi'$, $\kappa$) ← SatSolve($crits \cup (unkn \setminus CS) \cup atLeastOneIsTrue(CS)$)
7      if sat? then
8          $CS \leftarrow \{c \in CS \mid \pi' \not\models c\}$
9          $\pi \leftarrow \pi'$
10     else
11         return $(CS, \pi)$
12 return $(CS, \pi)$
```

Finding a Constrained MCS. There are two state of the art algorithms for finding MCSES, CLD [17] and Relaxation Search [3]. Both can be modified to find an MCS in a particular subset of the input clauses. We tried Relaxation Search but found that an approach that is similar to CLD, but not identical, worked best for our purposes. The resulting Algorithm 3 finds an MCS of the union of its two input clause sets, *crits* and *unkn* that is constrained to be contained in *unkn*.

Initially a SAT test is performed on *crits*. If *crits* is *unsat*, then there is no correction subset contained in *unkn* so the algorithm returns **null**. Otherwise, we have a satisfying model π of *crits*. The set of clauses falsified by any model is always a correction subset, and for π this correction subset, *CS*, is contained in *unkn*. The algorithm makes *CS* minimal by a sequence of SAT calls, each one asking the SAT solver to find a new model that falsifies a proper subset of clauses from the previous model. At each iteration, *CS* is updated to be the reduced set of falsified clauses. This continues until a model cannot be found or *CS* is reduced down to one clause. If a model cannot be found this means that adding any clause of *CS* to $(crits \cup unkn) \setminus CS$ yields an unsatisfiable formula, i.e., *CS* is an MCS. If *CS* is reduced to one clause then that clause must be an MCS since $crits \cup unkn$ is unsatisfiable, and an invariant of the algorithm is that *CS* is always a correction set of $crits \cup unkn$.

The main difference between Algorithm 3 and CLD of [17] lies in the encoding of *atLeastOneIsTrue*(*CS*) constraint passed to the SAT solver. In CLD this constraint is encoded as one large clause that is the disjunction of all of the clauses in *CS*. π falsifies all clauses of *CS*, so it must falsify their disjunction, therefore this disjunction is not a tautology. Furthermore, when the disjunction is satisfied at least one more clause of *CS* must also be satisfied. In Algorithm 3 we instead add a **selection variable** to each clause of *CS*. That is, each clause $c_i \in CS$ is transformed into the clause $c_i^+ = c_i \vee \neg s_i$, where s_i is a new variable not appearing elsewhere in the formula. Making s_i true strengthens c_i^+ back to c_i, while making it false satisfies c_i^+, effectively removing it from the CNF. With these selector variables $\mathit{atLeastOneIsTrue}(CS)$ can be encoded as a clause containing all of the selector variables: $\bigvee_{c_i \in CS} s_i$.

In addition, we found that in 90 % of cases when the SAT call is able to find an improving model it was able to do so without backtracking (no conflicts were found). Hence, a lot of the time of the solver is spent in descending a single branch that changes little between iterations of Algorithm 3. This time can be significantly reduced if we backtrack only to the point where the branches diverge. This is similar to techniques used already in general SAT solving for improving restarts [21] and in incremental SAT solving for reducing the overhead of assumptions [1]. We found that these two simple changes had a surprisingly positive effect on efficiency.

Recursive Model Rotation (RMR). If $\mathcal{F}$ is unsatisfiable then it follows from the definition that a clause c is critical for $\mathcal{F}$ if and only if there exists a model π such that $\pi \models \mathcal{F} \setminus \{c\}$. Hence, if in Algorithm 1 or Algorithm 2, we find for the current set of working clauses $W = (\mathit{unkn} \cup \mathit{crits})$ a model π such that $\pi \models W \setminus \{c\}$ we know that c is critical for W.

The technique of RMR [7] is to examine models that differ from π by only one assignment to see if we can find a model that witnesses the criticality of a clause different from c whose status is still undetermined. This is accomplished by flipping π's assignments to the variables of c one by one. Each such flipped model satisfies c and can be checked to see if it falsifies only one other unknown clause. If such a clause c' is found, then c' is now known to be critical, and we can recursively flip the model that witnesses its criticality. Recursive model rotation has been found to be very effective in MUS finding, eliminating many SAT tests. *Eager* model rotation (ERMR) [20] improves RMR by allowing it to falsify a critical clause, which may enable further rotations.

We have found that we can effectively find more critical clauses than ERMR using Algorithm 4. This algorithm first runs ERMR and then uses a SAT solver to find a model that witnesses the criticality of a clause of unknown status. This is done by using a standard encoding of an "at most one" constraint over the negation of the selector variables of the clauses currently in *unkn*. This forces the model to satisfy all clauses of $\mathit{crits} \cup \mathit{unkn}$ except one. (The model must falsify that remaining clause as $\mathit{crits} \cup \mathit{unkn}$ is unsatisfiable). This sequence of SAT calls can in principle find all critical clauses, but it can sometimes take too long. In practice, we put a strict time bound on the SAT call, and found that within that bound we could still find a useful number of additional critical clauses. As we will show in Sect. 5, this method can sometimes hinder performance, but also allows us to solve some instances that were otherwise too hard.

4.2 Finding All MUSES

We have also developed an algorithm for finding all MUSES. Our algorithm exploits the idea of using a SAT formula to represent a constrained collection

Algorithm 4. **em-rmr**$(c, crits, unkn, \pi)$ find more criticals than ERMR

```
Input: $(c, crits, unkn, \pi)$: $crits \cup unkn$ is $unsat$; all clauses of $crits$ are critical
       for $crits \cup unkn$, $c \in crits$; $\pi \not\models c$, and $\pi \models (crits \cup unkn) \setminus \{c\}$.
Output: Returns an additional set of clauses critical for $crits \cup unkn$.
1  while $true$ do
2      $crits' \leftarrow crits \cup$ ERMR $(c, crits, unkn, \pi)$
3      $unkn' \leftarrow unkn \setminus crits'$
4      $(\text{sat?}, \pi, \kappa) \leftarrow \text{SatSolve}(crits' \cup atMostOne(\{\neg s_i \mid c_i \in unkn'\}))$
5      if $sat?$ then
6          $c \leftarrow$ the single $c_i \in unkn'$ such that $\pi \models \neg s_i$
7          $crits' \leftarrow crits' \cup \{c\}$
8          $unkn' \leftarrow unkn' \setminus \{c\}$
9      else
10         return $(crits' \setminus crits)$
```

of sets. This idea was also used in the MARCO system which also enumerates all MUSES [15]. Specifically, if we regard each propositional variable as being a set element, then the set of variables made true by any model can be viewed as being a set. The set of satisfying models then represents a collection of sets.

In MARCO, all MUSES are enumerated by maintaining a SAT formula *ClsSets* which contains a variable s_i for each clause $c_i \in \mathcal{F}$. Clauses are added to *ClsSets* to exclude all already found MUSES as well as all supersets of these MUSES. For example, if $M = \{c_1, c_3, c_4\}$ is a known MUS then the clause $(\neg s_1 \vee \neg s_3 \vee \neg s_4)$ ensures that every satisfying model of *ClsSets* excludes at least one clause of M—this blocks M and all supersets of M from being solutions to *ClsSets*. A SAT test is preformed on *ClsSets* which extracts a subset $\mathcal{F}'$ of $\mathcal{F}$ not containing any known MUS. If $\mathcal{F}'$ is *unsat* one of its MUSES is extracted using Algorithm 1 and then blocked in *ClsSets*, otherwise MARCO grows $\mathcal{F}'$ into an MSS-MCS pair $\langle S, \mathcal{F} \setminus S\rangle$ and a clause is added to *ClsSets* to block $\mathcal{F} \setminus S$ and all of its supersets. For example, for a correction subset $C = \{c_1, c_3, c_4\}$ the clause $(s_1 \vee s_3 \vee s_4)$ is added to *ClsSets*. When *ClsSets* becomes *unsat*, all MUSES have been enumerated.

Algorithm 5 is our new algorithm for enumerating all MUSES of an unsatisfiable formula $\mathcal{F}$. The high level structure of Algorithm 5 is similar to that of MARCO but rather than treating the MUS extraction procedure as a black box, it records the (not necessarily minimal) correction subsets discovered during the MUS procedure and uses them to accelerate the extraction of future MUSES. In particular, MUSES and MCSES are blocked in the same way as in MARCO. Hence, at any stage the set $\mathcal{F}'$ obtained from a SAT solution of *ClsSets* has the properties (a) $\mathcal{F}'$ does not contain any known MUSES and (b) $\mathcal{F}'$ hits all known correction subsets. We want $\mathcal{F}'$ to hit all known correction subsets as otherwise $\mathcal{F}'$ cannot contain a MUS. When *ClsSets* becomes unsatisfiable all MUSES have been enumerated (and blocked).

Given a SAT solution $\mathcal{F}'$ of *ClsSets*, we extract a MUS using a procedure similar to Algorithm 2. In addition, however, we mirror the removal of clauses from *unkn* by setting the corresponding variable in *ClsSets* to *false*. Unit propagation in *ClsSets* may then give us more variables that can be moved to *crits*, because

Algorithm 5. MCS-MUS-ALL($\mathcal{F}$): Enumerate all MUSES of $\mathcal{F}$ using MCS duality.

```
Input: $\mathcal{F}$ an unsatisfiable set of clauses
Output: enumerate all MUSES of $\mathcal{F}$
1  $ClsSets \leftarrow$ empty set of clauses and the set of variables $\{s_i \mid c_i \in \mathcal{F}\}$
2  while true do
3      $(sat?, \pi, \kappa) \leftarrow$ SatSolve($ClsSets$)          // Setting all decisions to true
4      if not sat? then
5          return                                                  // All MUSES enumerated
6      $\mathcal{F}' \leftarrow \{c_i \mid c_i \in \mathcal{F} \wedge \pi \models s_i\}$
7      $\mathcal{F}^c \leftarrow \mathcal{F} \setminus \mathcal{F}'$
8      $(sat?, \pi, \kappa) \leftarrow$ SatSolve($\mathcal{F}'$)
9      if sat? then
10         $ClsSets \leftarrow ClsSets \cup (\bigvee_{c_i \in \mathcal{F}^c} s_i)$ // MCS
11     else
12         $crits \leftarrow \{c_i \mid s_i \in UP(ClsSets \cup \{(\neg s_j) | c_j \in \mathcal{F}^c\}\}$
13         $unkn \leftarrow \mathcal{F}' \setminus crits$
14         while true do
15             $(CS, \pi) \leftarrow$ find-mcs($crits, unkn$)
16             if $CS = null$ then
17                 enumerate($crits$)                                  // crits is a MUS
18                 $ClsSets \leftarrow ClsSets \cup (\bigvee_{c_i \in crits} \neg s_i)$   // Block this MUS
19                 break
20             else
21                 $c \leftarrow$ choose $c \in CS$
22                 $CS^F \leftarrow$ extend-cs ($CS$, $\pi$, $\mathcal{F}'$, $\mathcal{F}^c$)
23                 $ClsSets \leftarrow ClsSets \cup (\bigvee_{c_i \in CS^F} s_i)$ // Correction set
24                 $\mathcal{F}^c \leftarrow \mathcal{F}^c \cup (CS \setminus \{c\})$
25                 $crits \leftarrow crits \cup \{c\} \cup \{c_i \mid s_i \in UP(ClsSets \cup \{(\neg s_j) | c_j \in \mathcal{F}^c\}\}$
26                 $unkn \leftarrow unkn \setminus (CS \cup crits)$
27                 $crits \leftarrow$ em-rmr ($c$,$crits$,$unkn$,$\pi$)
28                 $unkn \leftarrow$ refine-clause-set($crits$, $unkn$, $|CS| > 1$),
```

previously discovered correction sets must be hit. Once a MUS is constructed, all these assignments to *ClsSets* are retracted.

One complication that arises in comparison to Algorithm 2 is that when we discover an MCS, it is only an MCS of $crits \cup unkn$, but we can only add MCSES of $\mathcal{F}$ to *ClsSets*. Therefore, we need to extend each MCS that we discover to an MCS of $\mathcal{F}$. The function **extend-cs** does this by adding to CS all clauses of $\mathcal{F} \setminus (crits \cup unkn)$ that were violated by π. We choose not to minimize this CS, as it can be costly especially if $\mathcal{F}$ is much bigger than $crits \cup unkn$.

An additional insight arising from the ideas of Relaxation Search [3] is that if while solving *ClsSets* we force the SAT solver to always set its decision variables to *true*, then the set $\mathcal{F}'$ we obtain will be a *maximal* set satisfying (a) and (b) above. Thus the set of excluded clauses $\mathcal{F}^c = \mathcal{F} \setminus \mathcal{F}'$ must be a *minimal* hitting set of the set of known MUSES. Each known MUS in *ClsSets* forces the exclusion of at least one clause. Thus $\mathcal{F}^c$, is a hitting set of the known MUSES. Since setting all decision variables to *true* causes the inclusion of clauses, all exclusions must be forced by unit propagation. This means that each excluded clause arises from a MUS all of whose other clauses have already been included in $\mathcal{F}'$. That is, for each excluded clause c in $\mathcal{F}^c$ there is some known MUS M such that $M \cap \mathcal{F}^c = \{c\}$. This shows that $\mathcal{F}^c$ is minimal.

Theorem 2. *If its input formula $\mathcal{F}$ is unsat, All-MCS-MUS returns all MUSes of its input formula $\mathcal{F}$.*

Proof (Sketch). First note that all MUSES and all MSSES are solutions of *ClsSets*. At each iteration, it produces either a satisfiable set, whose complement is an MCS, or an unsatisfiable set which is reduced to a MUS. Each is subsequently blocked so cannot be reported again, nor can any of its supersets. Additionally, the inner loop generates correction subsets, which it blocks in *ClsSets*, without checking if they are MCSES of $\mathcal{F}$. If these are MCSES then they will not be produced as solutions of *ClsSets*. So the algorithm will produce only MUSES and MCSES before *ClsSets* becomes unsat. Additionally, it will produce all MUSES, as this is the only way to block such solutions.

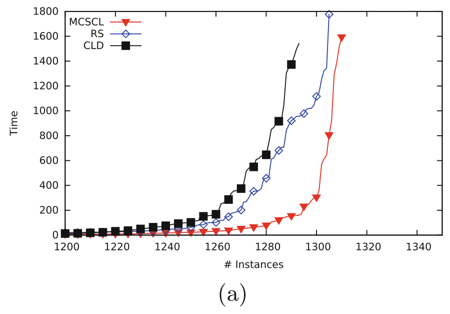
It remains to show that it produces correct MCSES and MUSES. For MCSES, it follows from the fact that the formula is satisfiable and the solution is maximal. For MUSES, we only need to show that unit propagation in *ClsSets* produces critical clauses. Indeed, all correction sets that are blocked in *ClsSets* are either produced by solutions of *ClsSets* itself or as MCSES of some subset of $\mathcal{F}$, extended to a correction set of $\mathcal{F}$. When such a blocking clause becomes unit, it means that exactly one of the clauses of the corresponding correction set remains in $crits \cup unkn$. A MUS must hit all correction sets, so the sole remaining clause is critical for $crits \cup unkn$. The correctness of the rest of the MUS extraction procedure follows from the correctness of Algorithm 2. ■

5 Empirical Evaluation

We implemented all of our new algorithms C++, and evaluated them against state-of-the-art algorithms for the corresponding tasks. We ran all experiments on a cluster of 48-core Opteron 6176 nodes at 2.3 GHz having 378 GB RAM.

Discovering MCSES. The state of the art in discovering MCSES is CLD [17] and Relaxation Search (RS) [3]. We compared Algorithm 3, (denoted MCSCL) using MINISAT as the *sat* solver without preprocessing [10], against CLD and RS in the tasks of identifying a single MCS and generating all MCSES of a formula, on a set comprising 1343 industrial instances from SAT competitions and structured MAXSAT instances from MAXSAT evaluations [17]. We show cactus plots comparing the three algorithms in both tasks in Fig. 1, with a timeout of 1800 seconds.

We first note that there is a relatively small window in which the algorithms may differentiate. In the case of discovering a single MCS, more than 1200 instances are solved instantaneously by all 3 algorithms, while some 20 of



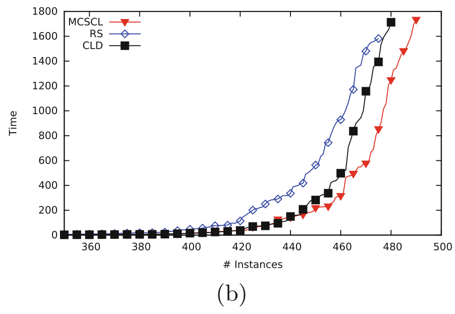


Fig. 1. Number of solved instances against time for (a) generating a single MCS and (b) generating all MCSES of a formula.

them are out of reach for all. Regardless, MCSCL is faster than the other two algorithms, for easy instances as well as hard ones and finds an MCS in 17 more instances than CLD and 5 more instances than RS. Similarly, in the case of discovering all MCSES, all 3 algorithms solve approximately 400 instances in less than a second, while 700 have too many MCSES to enumerate. In this case, MCSCL again outperforms both the other alternatives, finding all MCSES in 15 more instances than RS and 9 more instances than CLD.

Discovering a Single MUS. For this task, we used a set of 324 instances assembled by Belov et al. [5] for the evaluation of the tool DMUSER. We tested implementations of MCS-MUS that used either MINISAT or GLUCOSE [2] as the backend *sat* solver both with preprocessing enabled. We modified these solvers to bound time spent in preprocessing to 5 % of total runtime. We evaluated MCS-MUS with **em-rmr** or with only eager model rotation. We compared against MUSER [8] using MINISAT and using GLUCOSE, and against HAIFA-MUC [20] (based on MINISAT). For all algorithms, we preprocess the instances by *trimming* them using GLUCOSE and the tool DRAT-TRIM, which is a particularly effective heuristic clause set refinement method, but which cannot prove minimality and rarely produces a minimal MUS. We also compare against DMUSER [5] a system that augments a "core" MUS extraction algorithms with more elaborate trimming techniques. DMUSER yields significant improvements to MUSER and HAIFA-MUC and potentially could also improve MCS-MUS. However, we have not, as yet, integrated MCS-MUS into DMUSER to test this. The timeout in this experiment—including trimming—was set to 3600 seconds. Results are shown in Fig. 2.

Our first observation is that the combination of MINISAT and assumption-based solving is deficient, as is evident by the very poor performance of MUSER with MINISAT. Nevertheless, MCS-MUS with MINISAT is comparable to both HAIFA-MUC and MUSER with glucose[1]. We also see that **em-rmr** improves performance

[1] Our results seem to contradict the findings of Belov et al. [5], who found that MUSER with GLUCOSE was worse than HAIFA-MUC. It is unclear why this is the case.

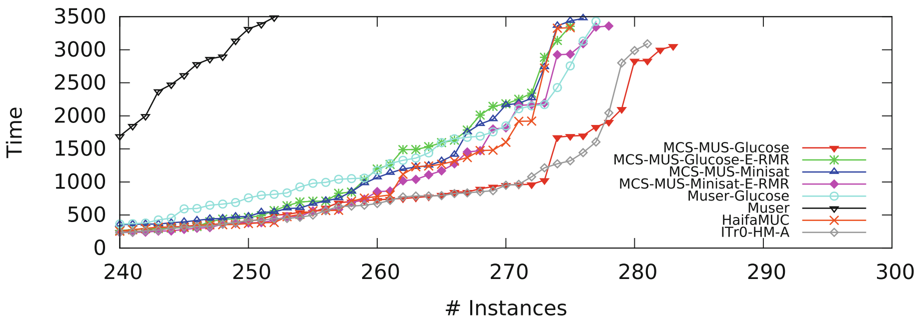


Fig. 2. Number of solved instances against time for extracting a single MUS.

overall in this case, yielding the second best combination among core algorithms. When paired with its GLUCOSE backend, MCS-MUS becomes the overall best algorithm for this task, surpassing even ITr-HM-A, the best configuration of DMUSER reported in [5]. However, the improvement of MCS-MUS when moving from MINISAT to GLUCOSE is not as dramatic as that of MUSER. It is, however, clearly ahead of other core algorithms and although it solves just 6 more instances than the next closest algorithm it does so significantly faster and even solves 2 more instances than DMUSER. Interestingly, **em-rmr** improves MCS-MUS with MINISAT but makes things worse when GLUCOSE is used.

Discovering All MUSES. Here we compare Algorithm 5 (MCS-MUS-ALL) against the state of the art for discovering multiple (potentially all) MUSES, MARCO [15]. We use only the MINISAT backend, as that is what MARCO is based on, with the additional optimization that for every unsatisfiable subset that we minimize, we create a copy of the SAT solver in which we can do destructive updates. This is implicit in MARCO, which uses an external tool for extracting a MUS.

We used the set of benchmarks from the MUS track of the 2011 SAT competition[2] (without trimming) and ran both algorithms for 3600 seconds on each instance. In Fig. 3 we show scatter plots of the time taken by each algorithm to generate the first MUS, of the time taken to differentiate between an instance with one MUS or many and of the number of MUSES generated within the timeout. Determining whether an instance contains one MUS or many involves either successfully terminating generation or generating a second MUS.

We see that MCS-MUS-ALL is more effective than MARCO at generating the first MUS and differentiating between instances with a single MUS or many MUSES. Indeed, it finds a MUS in 20 more instances than MARCO and differentiates 17 more instances. However, when MARCO can generate several MUSES, it is typically more efficient at doing so, especially for very large numbers of MUSES. We conjecture that in these cases, extracting a single MUS is so efficient, that the

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

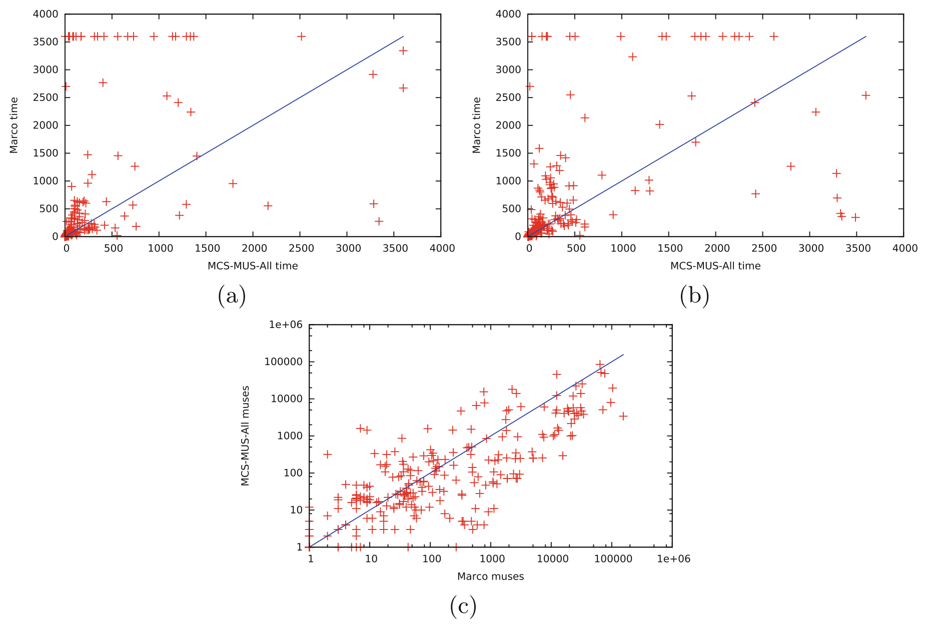


Fig. 3. (a) Time for finding a single MUS, (b) Time to differentiate between single-MUS and multiple-MUS instances, (c) number of MUSES generated, *in logscale*. In all cases, points above the line indicate MCS-MUS-ALL was better.

overhead of keeping track of the correction sets that MCS-MUS-ALL generates outweighs their potential benefit. This means that when the objective is to generate a variety of MUSES quickly on instances of moderate difficulty, MCS-MUS-ALL is to be preferred, but for large numbers of MUSES in easy instances, MARCO is preferable.

6 Conclusions

We have proposed a novel approach to extracting MUSES from unsatisfiable formulas. We exploited the well-known hitting set duality between correction sets and unsatisfiable subsets and used a greedy approach which, given an unhit MCS, can extend a set of clauses so that they are guaranteed to be a subset of a MUS. We further extended this algorithm to generating all MUSES. These developments hinge in part on our new very efficient MCS extraction algorithm. In all cases, we have demonstrated that the new algorithms outperform the state of the art. Despite this, there is little tuning or low level optimizations in our implementation, in contrast to the current state of the art [20]. This suggests that in future work we explore such optimizations to widen the gap.

References

1. Audemard, G., Lagniez, J.-M., Simon, L.: Improving glucose for incremental SAT solving with assumptions: application to MUS extraction. In: Järvisalo, M., Van Gelder, A. (eds.) SAT 2013. LNCS, vol. 7962, pp. 309–317. Springer, Heidelberg (2013)
2. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI), pp. 399–404 (2009)
3. Bacchus, F., Davies, J., Tsimpoukelli, M., Katsirelos, G.: Relaxation search: a simple way of managing optional clauses. In: Proceedings of the AAAI National Conference (AAAI), pp. 835–841 (2014)
4. Belov, A., Chen, H., Mishchenko, A., Marques-Silva, J.: Core minimization in sat-based abstraction. In: Design, Automation and Test in Europe (DATE), pp. 1411–1416 (2013)
5. Belov, A., Heule, M.J.H., Marques-Silva, J.: MUS extraction using clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 48–57. Springer, Heidelberg (2014)
6. Belov, A., Lynce, I., Marques-Silva, J.: Towards efficient MUS extraction. AI Commun. **25**(2), 97–116 (2012)
7. Belov, A., Marques-Silva, J.: Accelerating MUS extraction with recursive model rotation. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 37–40 (2011)
8. Belov, A., Marques-Silva, J.: Muser2: an efficient MUS extractor. JSAT **8**(1/2), 123–128 (2012)
9. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010)
10. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
11. Grégoire, É., Mazure, B., Piette, C.: On approaches to explaining infeasibility of sets of boolean clauses. In: International Conference on Tools with Artificial Intelligence (ICTAI), pp. 74–83 (2008)
12. Gurfinkel, A., Belov, A.: FrankenBit: bit-precise verification with many bits. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 408–411. Springer, Heidelberg (2014)
13. Gurfinkel, A., Belov, A., Marques-Silva, J.: Synthesizing Safe Bit-Precise Invariants. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 93–108. Springer, Heidelberg (2014)
14. Lagniez, J.-M., Biere, A.: Factoring out assumptions to speed up MUS extraction. In: Järvisalo, M., Van Gelder, A. (eds.) SAT 2013. LNCS, vol. 7962, pp. 276–292. Springer, Heidelberg (2013)
15. Liffiton, M.H., Malik, A.: Enumerating infeasibility: finding multiple MUSes quickly. In: Gomes, C., Sellmann, M. (eds.) CPAIOR 2013. LNCS, vol. 7874, pp. 160–175. Springer, Heidelberg (2013)
16. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reasoning **40**(1), 1–33 (2008)
17. Marques-Silva, J., Heras, F., Janota, M., Previti, A., Belov, A.: On computing minimal correction subsets. In: Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI), pp. 615–622 (2013)

18. Marques-Silva, J., Janota, M., Belov, A.: Minimal sets over monotone predicates in boolean formulae. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 592–607. Springer, Heidelberg (2013)
19. Nadel, A., Ryvchin, V., Strichman, O.: Efficient MUS extraction with resolution. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 197–200 (2013)
20. Nadel, A., Ryvchin, V., Strichman, O.: Accelerated deletion-based extraction of minimal unsatisfiable cores. J. Satisfiability Boolean Model. Comput. (JSAT) **9**, 27–51 (2014)
21. Ramos, A., van der Tak, P., Heule, M.J.H.: Between restarts and backjumps. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 216–229. Springer, Heidelberg (2011)
22. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. **32**(1), 57–95 (1987)
23. Torlak, E., Chang, F.S.-H., Jackson, D.: Finding minimal unsatisfiable cores of declarative specifications. In: Proceedings of the International Symposium on Formal Methods (FM), pp. 326–341 (2008)
24. Wieringa, S.: Understanding, improving and parallelizing mus finding using model rotation. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 672–687. Springer, Heidelberg (2012)