Towards Efficient MUS Extraction

Anton Belov [a], Inês Lynce [b] and
Joao Marques-Silva [a,b,*]
[a] *CSI/CASL*
University College Dublin, Ireland
E-mail: {anton.belov,jpms}@ucd.ie
[b] *IST/INESC-ID*
Technical University of Lisbon
E-mail: ines@sat.inesc-id.pf

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. Among these, this paper analyzes in detail the technique of recursive model rotation, which provides significant performance gains in practice. Experimental results, obtained on representative practical benchmarks, indicate that the new algorithms achieve significant performance gains with respect to state of the art MUS extraction algorithms.

Keywords: Boolean Satisfiability, Minimally Unsatisfiable Subformula

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. [52,30,8,27, 26,19,20,21,51,22,16,23,39,44,48,40,4,5,38]). Most of this work is inspired by earlier work on computing explanations for inconsistencies (e.g. [14,10,3]). Algorithms for MUS extraction have often been characterized as *constructive* [22] (also referred to as insertion-based [16,39]), as *destructive* [22] (also referred to as removal-based [16], or deletion-based [39]), or as *dichotomic* [30,26]. 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 [22]. 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 [16], 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. [39,38]). Despite the large body of work, MUS extraction algorithms are *not* industrial-strength, meaning that, with a few recent exceptions (e.g. [44]), 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 previous MUS extraction algorithms are shown to be in general inefficient for large complex problem instances from practical applications.

This paper extends recent work on developing industrial-strength MUS extraction algorithms [40,4], and its main contributions can be summarized as follows. 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

*Corresponding author: J. Marques-Silva, UCD CASL, Ireland.

AI Communications
ISSN 0921-7126, IOS Press.

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. Among the new techniques, the novel technique of *model rotation*, first proposed in [40] and further extended in [4], is shown to enable significant savings in terms of the SAT solver calls necessary for computing an MUS. 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. Compared to earlier work [40,4], this paper extends the analysis of model rotation, and identifies some of its limitations. In addition, this paper provides a more extensive experimental evaluation.

The rest of this paper is structured as follows ...

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. [18,41]). 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* [14], *destructive* [10,3] and *dichotomic* [30,26]. 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 [6,44] or constructive [51].

Most practical MUS computation algorithms iteratively identify *transition clauses* [22]. 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, $\texttt{SAT}(\mathcal{F})$ 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, $\texttt{SAT}(\mathcal{F})$ may also return the satisfying assignment and an unsatisfiable subset. In this case, the output of the SAT solver call is represented as follows: $(\mathsf{st}, \nu, \mathcal{U}) \leftarrow \texttt{SAT}(\mathcal{F})$. 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 [16].

The standard organization of a destructive MUS extraction algorithm is shown in Algorithm 1 [22,39]. 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 [22,16,39].

Algorithm 1: Destructive MUS Extraction

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

3. Applications of MUSes

Minimally unsatisfiable subsets of CNF formulas are used in a wide variety of contexts of theoretical and applied computer science. Some of the practical applications from the early 2000's that motivated the interest in algorithms for computing MUSes include type debugging in programming languages [50], circuit error diagnosis [25], and error localization in automotive product configuration data [49]. However, by the late 2000's it became clear that some of the technologies that traditionally relied on the computation of non-minimal unsatisfiable subsets of propositional formulas (e.g. [17,42,24,29]) – the *unsatisfiable cores* – can benefit significantly, and are willing to pay the price, for the computation of MUSes.

In this section we describe on a high level two, and go into details of another one of the recent applications of MUSes. These applications come from the domain of Computer Aided Design (CAD): formal equivalence checking, hardware model checking and logic synthesis.

In this section we have no choice but to assume that the reader is familiar with the basic terminology and some of the technologies used in CAD. If this is not the case, the section can be safely skipped.

3.1. Formal Equivalence Checking

Formal Equivalence Checking (FEC) [28] is a technique for formally proving equivalence of two design models. FEC is used in various stages of the VLSI design flow, for example in functional equivalence comparison of the golden Register Transfer Level (RTL) model against the implementation which might be created manually or by an automatic synthesis tool.

Due to the limited capacity of the formal verification engines, FEC has to be performed compositionally: the compared models are separated into small parts, the *slices*, and the equivalence between the slices is checked with BDD, or, more recently, SAT-based FEC engine. Note that any slice in isolation can have more behaviours than when it is part of the complete model – for example, some combinations of the input signals of the slice may be unrealizable in the complete model. As such, the FEC is performed with respect with the *environmental assumptions* which mimic the essential behaviour of the complete model with respect to the slice.

SAT-based FEC is performed by constructing a propositional formula that captures the logic of the two slices and the environmental assumptions. The constructed formula is unsatisfiable if and only if the slices, under the given assumptions, are functionally equivalent. If the equivalence of two slices is established, the assumptions need to be confirmed – that is the designer must prove that the assumptions are guaranteed by the model (this is an example of so-called *assume-guarantee* reasoning for compositional verification [1]). As such it is critical to reduce the number of assumptions required to prove the equivalence of the slices.

One of the approaches to the reduction of the number of the environmental assumption is to consider the unsatisfiable core of the unsatisfiable formula that establishes the equivalence between slices – any assumption that is not part of the core can be ignored. However, in practice the unsatisfiable cores produced by SAT solvers still contain a large number of assumptions [12]. Ideally, a smallest possible core is needed. However, the computation of a smallest core is extremely costly, and so MUSes provide an effective and practically feasible alternative. In [12] it was shown that MUS-based reduction of environmental assumptions in FEC a critical impact on the efficiency of the design flow.

3.2. Proof-based Abstraction Refinement

Proof-based abstraction refinement (PBA) [42] is a popular approach to model checking of large industrial hardware designs. The goal of model checking [11,47] is to establish correctness properties of finite state transition systems, which in the case of hardware, capture the Finite State Machine (FSM) of a hardware design.

The basic idea of PBA is to start with a Bounded Model Checking (BMC) run for some small depth k. In BMC [7] a propositional formula $BMC(k)$ is constructed in such a way that it is unsatisfiable if and only if no execution of the FSM with k or less steps violates the correctness property. If the formula $BMC(k)$ is satisfiable, the property is violated, and we are done. Otherwise, the unsatisfiable core of the formula $BMC(k)$ is used to construct an abstraction of the given design in the following way: for a latch L in the design, let $LC(L,k)$ be the set of clauses in the $BMC(k)$ that represent the input-output correspondence of the latch values on the execution steps. Then, if none of the clauses of $LC(L,k)$ partake in the unsatisfiable core of $BMC(k)$, the latch L is removed from the design (i.e. it is replaced with a primary input). The design abstracted in this way has more behaviours than the original one, however it still has the property that no executions of length k or less violate the correctness condition. The abstracted design is then checked with a complete (for example, BDD-based) model checker, and if the property is proved on the abstract design, it is guaranteed to hold on the concrete design. Otherwise, the length of the violating execution, which is provided by the complete model checker and is guaranteed to be larger than k, is used for the next BMC run.

Notice that each latch abstracted from the concrete design reduces the state space of the design by a factor of 2. Thus, it is extremely beneficial to abstract away as many latches as possible. As with our example of FEC, the smallest core of the formula $BMC(k)$ would be ideal for this purpose, however since it is too expensive to compute, an MUS is computed instead, and can be used effectively to eliminate additional latches. As suggested in [45] this procedure can be further optimized by computing the core terms of sets of clauses.

3.3. Boolean Function Bi-decomposition

Boolean function decomposition [2,13] is a fundamental operation in logic synthesis. Given a Boolean function $f(X)$ the task is to represent f in the form

$$f(X) = h(g_1(X), \ldots, g_m(X)),$$

such that that h and g_i's are simpler Boolean functions. Decomposition with $m = 2$ is referred to as *bi-decomposition*, and is of particular practical relevance due to the fact that logic netlists are most often expressed in terms of binary gates. To showcase the application of MUSes in this setting we now set up the necessary background.

Given a set of variables X, a *partition* of X is a set of pair-wise disjoint sets X_A, X_B, X_C such that $X = X_A \cup X_B \cup X_C$. A partition is *non-trivial* if $X_A \neq \emptyset$ and $X_B \neq \emptyset$. A partition is *disjoint* if $X_C = \emptyset$, and is *balanced* if $|X_A| = |X_B|$. Given a Boolean function $f(X)$ the *bi-decomposition* of f consists of a partition of X and two Boolean functions f_A and f_B such that

$$f(X) = f_A(X_A, X_C) \circ f_B(X_B, X_C), \qquad (1)$$

where $\circ$ is usually one of $\vee$, $\wedge$, or $\oplus$.

The primary issue in bi-decomposition is to obtain a good partition of variables, i.e. a partition that is non-trivial, almost balanced, and such that the set of common variables X_C is small (or even empty) – the latter condition is the most important due to the fact that it affects directly the amount of wiring in the synthesized circuit. Once the partition is known, the functions f_A and f_B can be computed using by various methods, for example using BDDs or SAT and Craig interpolation (cf. [43,33]).

In [33,9] the authors show that the problem of the existence of a particular non-trivial partition can be reduced to checking the unsatisfiability of a certain propositional formula. Furthermore, the unsatisfiable cores of this formula correspond to other non-trivial partitions such that the size of the core is related directly to the quality of the partitions. For example, for the case of OR bi-decomposition (i.e. when $\circ = \vee$ in (1)), the aforementioned formula is given by following proposition.

Proposition 1 (cf. Proposition 2, [9]) *Let X_A, X_B, X_C be a non-trivial partition of the set of variables of a Boolean function $f(X)$. Then, f can be decomposed into $f_A(X_A, X_C) \vee f_B(X_B, X_c)$ for some functions f_A, f_B if and only if the following propositional formula $\mathcal{F}$ is unsatisfiable:*

$$\mathcal{F} = f(X) \wedge \neg f(X') \wedge \neg f(X'') \wedge \mathcal{F}_A \wedge \mathcal{F}_B, \quad (2)$$

where

(i) X', X'' *are the sets of primed versions of variables in* X*;*

(ii) $\mathcal{F}_A = \bigwedge_{x \in X_B \cup X_C}(x \equiv x')$*;*

(iii) $\mathcal{F}_B = \bigwedge_{x \in X_A \cup X_C}(x \equiv x'')$.

Example 1 *Let* $X = \{p, q, r\}$, $X_A = \{p\}$, $X_B = \{q\}$, $X_C = \{r\}$. *Then a function* $f(p, q, r)$ *can be represented as* $f_A(p, r) \vee f_B(q, r)$ *if and only if the propositional formula*

$$f(p,q,r) \wedge \neg f(p',q',r') \wedge \neg f(p'',q'',r'') \wedge$$
$$(q \equiv q') \wedge (r \equiv r') \wedge (p \equiv p'') \wedge (r \equiv r'')$$

is unsatisfiable.

Using Proposition 1 variable partitions are computed in the following way. The computation begins by identifying a non-trivial *seed* partition of X – such partition can for example be constructed by selecting two variables $x_i, x_j \in X$ and setting $X_A = \{x_i\}, X_B = \{x_j\}, X_C = X \setminus \{x_i, x_j\}$, and verifying the unsatisfiability of the formula (2). If the formula is unsatisfiable, a seed partition is found, otherwise another pair of variables is selected. The selection of variables for the seed partition can be aided by heuristics (e.g. [9]).

Note that the quality of the seed partition is rather poor – most of the variables are in the common set X_C. However, the quality of the partition can now be improved by considering the unsatisfiable cores of the formula (2): if for some variable $x \in X$ a core contains only the clauses that correspond to $(x \equiv x')$ (resp. $(x \equiv x'')$) then the variable x can be moved to the set X_B (resp. X_A). If the core doesn't contain either of these clauses, the variable x can be moved either to X_A or X_B (this way the partition can be made more balanced).

Clearly, small unsatisfiable cores are likely to correspond to good partitions, hence the minimization of the core size is of the key importance in this application. Since, again, the computation of the smallest unsatisfiable core is too expensive, MUSes provide an effective and efficient alternative. The results reported in [9] demonstrate that the MUS-based variable partitions are of significantly better quality that those based on non-minimal unsatisfiable cores.

4. 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 [22,39,38].

Algorithm 2 shows the new constructive MUS extraction algorithm. This new algorithm borrows ideas from a number of earlier algorithms. Similarly to AMUSE [46], it adds relaxation variables to all clauses. In addition, and similarly to the use of weighted MaxSAT for MUS extraction [16], 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. □

Algorithm 2: Constructive MUS Extraction with AtMost1 Constraint

Input : Unsatisfiable CNF Formula $\mathcal{F}$
Output: MUS $\mathcal{M}$

1 **begin**
2 $\quad\mathcal{M} \leftarrow \emptyset$ // $\mathcal{M}$: MUS under-approximation
3 $\quad\mathcal{R} \leftarrow \{r_i \,|\, r_i \text{ is fresh variable for } c_i \in \mathcal{F}\}$ // $\mathcal{R}$: relaxation variables
4 $\quad\mathcal{F}^R \leftarrow \{c_i \cup \{r_i\} \,|\, r_i \in R \wedge c_i \in \mathcal{F}\}$ // $\mathcal{F}^R$: working formula
5 $\quad\mathcal{T} \leftarrow \texttt{CNF}(\sum_{r_i \in R} r_i \leq 1)$ // ≤ 1 constraint
6 $\quad$**while** $\mathcal{F}^R \neq \emptyset$ **do** // Repeat while relaxed clauses exist
7 $\quad\quad(\mathsf{st}, \nu, \mathcal{U}) \leftarrow \texttt{SAT}(\mathcal{F}^R \cup \mathcal{T} \cup \mathcal{M})$
8 $\quad\quad$**if** st = true **then**
9 $\quad\quad\quad r_i \leftarrow \texttt{TrueVariable}(\nu, R)$ // Get true relaxation variable
10 $\quad\quad\quad c_i^R \leftarrow \texttt{Clause}(\mathcal{F}^R, r_i)$ // Get clause associated with r_i
11 $\quad\quad\quad \mathcal{F}^R \leftarrow \mathcal{F}^R \setminus \{c_i^R\}$ // Remove clause $c_i^R = c_i \cup \{r_i\}$ from $\mathcal{F}^R$
12 $\quad\quad\quad \mathcal{M} \leftarrow \mathcal{M} \cup \{c_i^R \setminus \{r_i\}\}$ // Add clause $c_i = c_i^R \setminus \{r_i\}$ to MUS
13 $\quad\quad$**else** // If unsatisfiable, $\mathcal{U} \cap \mathcal{T} \neq \emptyset$
14 $\quad\quad\quad$**if** $\mathcal{U} \cap \mathcal{F}^R = \emptyset$ **then**
15 $\quad\quad\quad\quad \mathcal{F}^R \leftarrow \emptyset$
16 $\quad\quad\quad$**else**
17 $\quad\quad\quad\quad c_i^R \leftarrow \texttt{SelectClause}(\mathcal{F}^R \cap \mathcal{U})$
18 $\quad\quad\quad\quad \mathcal{F}^R \leftarrow \mathcal{F}^R \setminus \{c_i^R\}$ // Block one MUS
19 $\quad$**return** $\mathcal{M}$ // Final $\mathcal{M}$ is an MUS
20 **end**

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.

5. 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.

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. 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.

5.1. Clause-Set Trimming

A standard preprocessing technique for computing MUSes of large CNF formulas is *clause set trimming*. Trimming consists of iteratively calling a SAT solver on computed unsatisfiable subformulas until no changes are detected in between calls to the SAT solver [52]. Nevertheless, 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 next.

5.2. Clause-Set Refinement

Next, we analyze the technique summarized in line 12 of Algorithm 3.

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 $\mathcal{U} \setminus \mathcal{M}$.*

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 [15,44]. 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).

Algorithm 3: Hybrid MUS Extraction

```
   Input  : (Trimmed) Unsatisfiable CNF Formula F
   Output: MUS M
1  begin
2      F' ← F                                          // Working CNF formula
3      M ← ∅                                           // MUS under-approximation
4      while F' ≠ ∅ do
5          c_i ← GetClause(F')
6          F' ← F' \ {c_i}
7          (st, ν, U) = SAT(M ∪ F' ∪ {¬c_i})           // Add redundancy checking
8          if st = true then                           // If SAT, c_i is transition clause
9              M ← M ∪ {c_i}
10             RMR(F' ∪ M, M, ν)                       // Recursive model rotation
11         else if U ⊆ M ∪ F' then                     // Equivalently, if U ∩ {¬c_i} = ∅
12             F' ← U \ M                              // Clause-set refinement
13     return M                                        // Final M is an MUS
14 end
```

5.3. Redundancy Removal

The redundancy removal technique consists of constraining the SAT solver call, as shown in line 7 of Algorithm 3. The additional constraints consists of 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 [34]. Although this technique was first used in [51], in the context of a constructive MUS extraction algorithm involving $\mathcal{O}(m \times k)$ calls to a SAT solver, 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.

The integration of the redundancy removal technique (line 7) and clause set refinement is not immediate, since the clauses from the redundancy removal technique can be part of the computed unsatisfiable core. The solution is to include a test (line 11 of Algorithm 3) to decide when the unsatisfiable core can be used as the next working CNF formula.

Proposition 2 *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 $\mathcal{F}$.*

5.4. Recursive Model Rotation

Finally, we describe 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 from $\mathcal{F}'$ and satisfies all clauses in $\mathcal{M}$ *proves* that c 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 the 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. This is illustrated with the following example.

For clarity of the presentation we introduce the following notation: given a CNF formula $\mathcal{F}$ and an assignment ν, $Unsat(\mathcal{F}, \nu)$ denotes the set of clauses in $\mathcal{F}$ falsified by ν. The expressions $Var(c)$ (resp. $Var(\mathcal{S})$) denote the set of variables that occur in the clause c (resp. set of clauses $\mathcal{S}$). Finally, given an assignment ν, $\nu|_{\neg x}$ denotes the assignment that agrees with ν on all variables except x.

Example 2 (Model Rotation) *Let* $\mathcal{F} = \{c_1, \ldots, c_5\}$ *be an unsatisfiable formula with*

$$c_1 = \{\neg x_1, \neg x_2\} \quad c_3 = \{x_2, \neg x_3\} \quad c_5 = \{x_1, x_2\}$$
$$c_2 = \{x_1, \neg x_2\} \quad c_4 = \{x_2, x_3\}$$

Suppose that c_1 *is removed from* $\mathcal{F}$*. Then* $\mathcal{F} \setminus \{c_1\}$ *is satisfiable, and let* $\nu_1 = \{x_1, x_2, x_3\}$ *be the model of* $\mathcal{F} \setminus \{c_1\}$ *returned by the SAT solver. We have* $Unsat(\mathcal{F}, \nu_1) = \{c_1\}$*. Let* $\nu_2 = \nu_1|_{\neg x_1}$*, that is* $\nu_2 = \{\neg x_1, x_2, x_3\}$*. We now have* $Unsat(\mathcal{F}, \nu_2) = \{c_2\}$*, and therefore,* c_2 *is another transition clause of* $\mathcal{F}$*.*

The process can be continued until for some clause c and the corresponding model ν of $\mathcal{F} \setminus \{c\}$, for every variable $x \in Var(c)$ the set $Unsat(\mathcal{F}, \nu|_{\neg x})$ is either not a singleton, or contains a clause that is already known to be a transition clause. In the above example, the rotation stops at ν_2 as $Unsat(\mathcal{F}, \nu_2|_{\neg x_2}) = \{c_3, c_5\}$, while $Unsat(\mathcal{F}, \nu_2|_{\neg x_1}) = \{c_1\}$[1]. Note, however, that since the clause c_1 has two literals, we can "backtrack" to the initial assignment ν_1 and attempt to flip the variable x_2. This is motivated by the following observation.

Lemma 7 *Let* $\mathcal{F}$ *be an unsatisfiable formula, let* $c \in \mathcal{F}$ *be a transition clause, and let* ν *be a model of* $\mathcal{F} \setminus \{c\}$*. Then, the sets* $Unsat(\mathcal{F}, \nu|_{\neg x})$ *for* $x \in Var(c)$ *are pairwise disjoint.*

Proof. Take $x \in Var(c)$, and let c' be some clause in $Unsat(\mathcal{F}, \nu|_{\neg x})$. Since $c' \notin Unsat(\mathcal{F}, \nu)$, the literal of variable x was *critical* in c' under ν (that is, the only literal in c' that evaluates to 1 under ν). Since every clause has *at most* one critical literal, the lemma follows. □

Hence, by performing model rotation on different variables of c we are *guaranteed* to obtain disjoint sets of clauses, thus increasing the likelihood of detecting additional transition clauses.

Example 2 (continued) *We backtrack to the assignment* ν_1 *and flip variable* x_2 *to obtain the assignment* $\nu_3 = \{x_1, \neg x_2, x_3\}$*, and since* $Unsat(\mathcal{F}, \nu_3) = \{c_3\}$ *we have a new transition clause* c_3*. Rotation of* ν_3 *on variable* x_3 *results in the assignment* $\nu_4 = \{x_1, \neg x_2, \neg x_3\}$*, which gives another transition clause* c_4*. Rotating* ν_4 *on* x_2 *results in the assignment* $\nu_4 = \{x_1, x_2, \neg x_3\}$ *at which point the rotation terminates, because* $Unsat(\mathcal{F}, \nu_4) = \{c_1\}$ *and* c_1 *is already known to be a transition clause, and all possible rotations have been made.*

[1] $\nu_2|_{\neg x_1}$ is simply ν_1 — in the examples that follow we will omit the cases of "unflipping" variable.

In this example, such *recursive model rotation (RMR)* allows to detect all of the transition clauses of $\mathcal{F}$. Remarkably, as demonstrated in Section 7, the cases when RMR finds all, or close to all, of the transition clauses do occur often on practical instances.

The sketch of the algorithm for the recursive model rotation is presented in Algorithm 4. The algorithm is invoked whenever an MUS extractor detects a new transition clause as a result of a call to a SAT solver. We note that the total number of model rotations during the execution of an MUS computation algorithm on any formula $\mathcal{F}$ is at most $k \cdot c_{max}$, where k is the size of the largest MUS $\mathcal{M}$ of $\mathcal{F}$, and c_{max} is the maximum among the lengths of clauses in $\mathcal{M}$. On the other hand, each successful model rotation (i.e. the one that detects a new transition clause) saves a potentially expensive call to a SAT solver. Given that in practical instances the size of MUSes rarely exceeds a few tens of thousands of clauses, it is not surprising that model rotation often provides for significant performance gains. In Section 7 we demonstrate these gains empirically, and also investigate whether RMR can be improved to allow to detect more transition clauses.

Clearly, model rotation could use more elaborate approaches for finding assignments that falsify 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 [51]. However, model rotation is a fundamentally different technique. Whereas the approach in [51] 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.

5.5. Analysis of Other Techniques

Algorithm 3 integrates, adapts and extends several techniques proposed in earlier work. One additional technique could be considered, namely autarkies [31]. For example, autarkies have been successfully used in recent MUS enumeration algorithms [36]. 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 $\mathcal{F}$

Algorithm 4: Recursive Model Rotation (RMR)

Input : $\mathcal{F}$ — an unsatisfiable CNF formula
: $\mathcal{M} \subseteq \mathcal{F}$ — a set of transition clauses of $\mathcal{F}$
: ν — a model of $\mathcal{F} \setminus \{c\}$ for some $c \in \mathcal{M}$
Effect: $\mathcal{M}$ may contain additional transition clauses of $\mathcal{F}$

```
1 begin
2     c ← the single clause in Unsat(F, ν)
3     foreach x ∈ Var(c) do
4         ν' ← ν|_¬x
5         if Unsat(F, ν') = {c'} and c' ∉ M then
6             M ← M ∪ {c'}
7             RMR (F, M, ν')
8 end
```

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.

5.6. Interfacing SAT Solvers

In MUS extraction algorithms, SAT solvers can either be used in incremental or non-incremental mode (e.g. [6]). Recent experimental results suggest that incremental mode provides significant performance gains [51,44]. 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 $\mathcal{M}$.

6. Experimental Results

The algorithms described in the previous sections were implemented in the MUS extraction tool MUSer (MUS ExtratoR), built on top of the Picosat [6] SAT solver. Supported by existing experimental evidence [39,38], the incremental interface of Picosat was used. (Observe that other work [44] 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 4; the hybrid MUS extraction algorithm (*HYB*) described in section 5; a reference destructive algorithm (*DREF*); a reference constructive algorithm [14] (*CREF*); the recent constructive algorithm from [51] (*MUNSAT*); a recent local-search-guided destructive MUS extraction algorithm from [21] (*AOMUS*); a well-known MUS extractor from [52] (ZMIN); SAT4J [32] MUS extractor in linear constructive mode (*S4J_I*), in QuickXPlain [30] mode (*S4J_Q*), and in destructive mode (*S4J_D*). Finally, a destructive MUS extraction algorithm available in the Picosat distribution [6] (PMUS). As shown by the results below, fairly recent MUS extractors [21,51,16] 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 MUS track of the 2011 SAT Competition [2]. 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 (based on invoking the SAT solver a fixed (3) number of times) 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

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

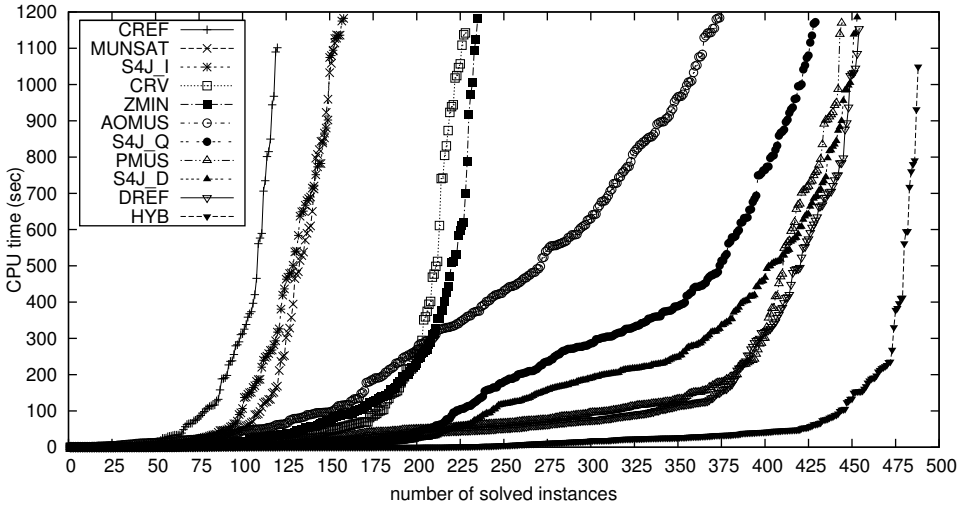

Fig. 1. Cactus plot with running times of MUS extractors.

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 [51] and AOMUS [21], perform 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 [21,51] 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 [44], on selected problem instances. The best run times from [44] 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.

7. More on Model Rotation

The goal of this section is to provide additional insights into the power and capabilities of recursive model rotation (RMR), as described in Section 5. This additional attention to the technique is justified by the analysis of the effect of RMR summarized in Fig. 3. The plot in Fig. 3, left demonstrates the impact of RMR on the runtime of HYB by comparing the run-

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	**488**

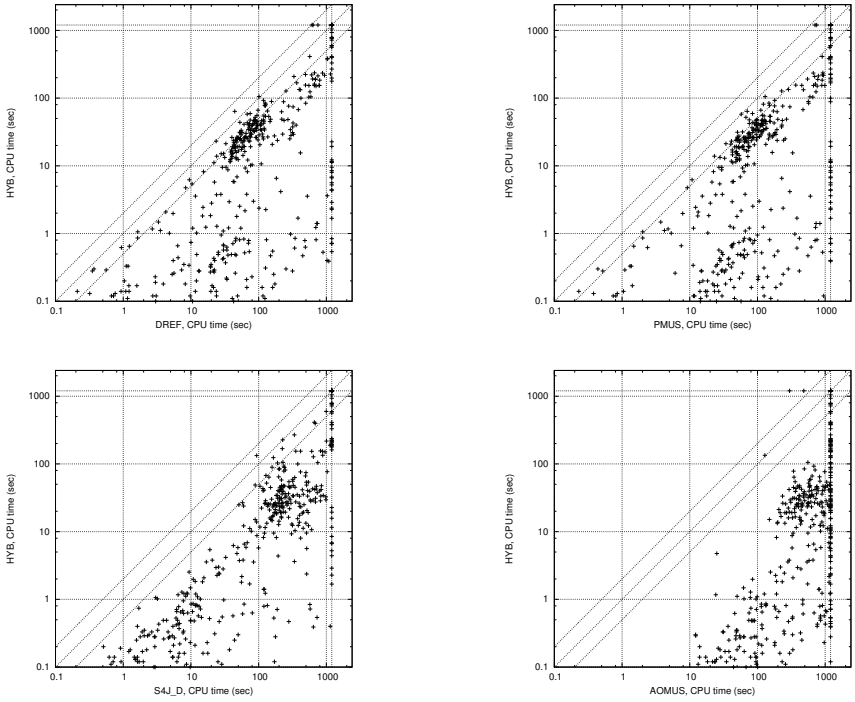

Fig. 2. Scatter plot comparing HYB with other MUS extractors: CPU time.

Table 2
Comparison with [44]

Instance	3pipe	4pipe_1	barrel6	barrel7	barrel8	longmult6	longmult7	longmult8
Best in [44]	**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	—

times of HYB and a version of HYB without RMR. We observe that RMR allows for significant, often multiple orders of magnitude, speed-ups in MUS extraction. Such speed-ups are explained by the significant reduction in the number of invocations of SAT solver during MUS extraction (Fig. 3, center) — in fact, we observe that in many cases MUS computation requires a single SAT solver invocation, which is the theoretical minimum[3]. Finally, we note that even in the cases when RMR cannot detect *all* of the transition clauses, the technique is still extremely effective – on vast majority of the benchmark instances RMR detects over 60% of all transition clauses (Fig. 3, right).

Given the apparent power of RMR, it is natural to ask whether the technique can be extended in a man-

[3]Recall that we assume that the input instance is unsatisfiable, hence we do not perform the first (UNSAT) call.

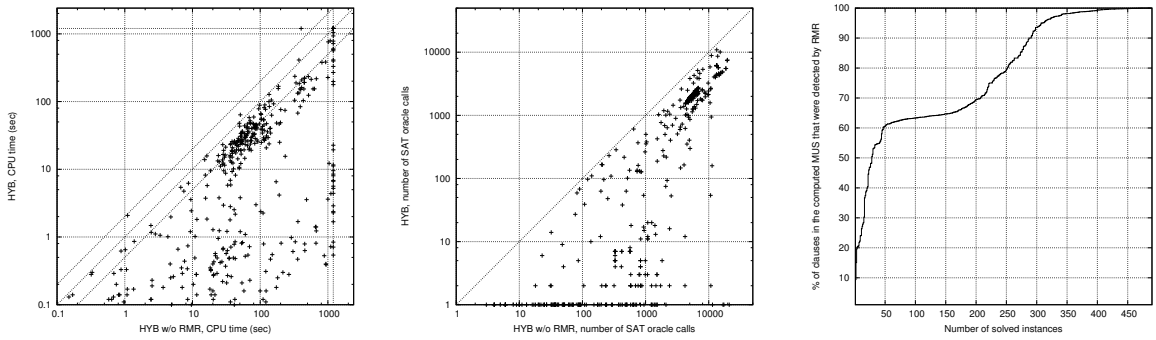

Fig. 3. Left: HYB vs HYB without RMR in terms of CPU time (sec). Center: HYB vs HYB without RMR in terms of the number of invocations of the SAT solver (solved instances only). Right: histogram of the percentage of clauses in the MUS computed by HYB detected using RMR.

ner that would allow to detect more, if not all, necessary clauses. It is also of interest to investigate whether some of the additional information can be extracted during the execution of the algorithm. In this section we put forward a number of proposals, and evaluate their effectiveness empirically.

We note that since an instance may not have any transition clauses (this happens when it has more than one disjoint MUS), RMR, or any of its possible extensions, can be only used as an optimization of MUS extraction algorithms, rather than a standalone technique for computation of MUSes. As such, any extension to RMR should maintain the low computational overhead of the technique.

To gain insight into the operation of RMR it is helpful to consider the following structure, which we call the *rotation graph* of an unsatisfiable CNF formula, defined as follows.

Definition 5 (Rotation graph) *Let $\mathcal{F}$ be an unsatisfiable CNF formula. The rotation graph of $\mathcal{F}$, in symbols $R_{\mathcal{F}}$, is a labelled directed graph $\langle V, E, L\rangle$, where*

(i) the set of vertices, V, is the set of all possible assignments to $Var(\mathcal{F})$;

(ii) each vertex $\nu \in V$ is labelled with the set $Unsat(\mathcal{F}, \nu)$ of clauses in $\mathcal{F}$ falsified by ν;

(iii) there is a directed edge $e = \langle \nu, \nu' \rangle \in E$, if $\nu' = \nu|_{\neg x}$ for some $x \in Unsat(\mathcal{F}, \nu)$.

Thus, the rotation graph $R_{\mathcal{F}}$ for a formula $\mathcal{F}$ has $2^{|Var(\mathcal{F})|}$ vertices that correspond to all possible assignments to variables of $\mathcal{F}$. Each vertex ν is labelled with the set $Unsat(\mathcal{F}, \nu)$. Each pair of assignments h and $\nu' = \nu|_{\neg x}$ on Hamming distance 1 from each other is connected by a directed edge $\langle \nu, \nu' \rangle$ if x is a variable that appears in a some clause falsified by ν.

Example 3 *Consider the formula $\mathcal{F}$ from Example 2 which we reproduce here for convenience. $\mathcal{F} = \{c_1, \ldots, c_5\}$, where*

$$c_1 = \{\neg x_1, \neg x_2\} \quad c_3 = \{x_2, \neg x_3\} \quad c_5 = \{x_1, x_2\}$$
$$c_2 = \{x_1, \neg x_2\} \quad c_4 = \{x_2, x_3\}$$

The rotation graph $R_{\mathcal{F}}$ is shown in Figure 4. Note that a pair of directed edges in opposite directions is depicted as double-headed arrow. The vertex (assignment) $\nu_1 = \{x_1, x_2, x_3\}$ is labelled with $\{c_1\}$, because c_1 is the only clause in $\mathcal{F}$ falsified by ν_1. There is a directed edge from ν_1 to $\nu_2 = \{\neg x_1, x_2, x_3\}$ because $x_1 \in Var(c_1)$. However, since $x_3 \notin Var(c_1)$, there is no edge from ν_1 to $\{x_1, x_2, \neg x_3\}$.

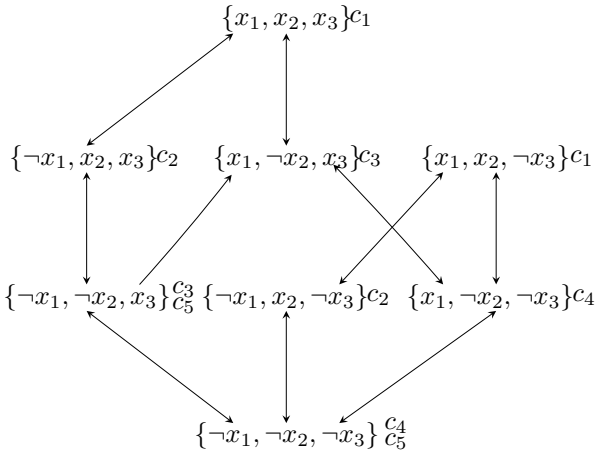

Fig. 4. Rotation graph for a formula $\mathcal{F}$ from Example 3. Double–headed arrows represent a pair of edges in the opposite directions. The label of a vertex is shown on the right side of it.

A rather straightforward observation is that for every transition clause $c \in \mathcal{F}$ there is a vertex ν in $R_{\mathcal{F}}$

labelled solely by c, i.e. $Unsat(\mathcal{F}, \nu) = \{c\}$. We will say that ν is a *distinguishing* assignment (or vertex) for c in this case.

Definition 6 (Distinguishing assignment) *Let $\mathcal{F}$ be an unsatisfiable CNF formula. An assignment ν is a* distinguishing assignment (for some clause c) *if $Unsat(\mathcal{F}, \nu) = \{c\}$.*

Thus, given the unsatisfiable formula $\mathcal{F}$, the set $\mathcal{M} \subseteq \mathcal{F}$ that contains some of the necessary clauses of $\mathcal{F}$ and a distinguishing assignment ν for some $c \in \mathcal{M}$, the execution of RMR($\mathcal{F}, \mathcal{M}, \nu$) (see Algorithm 4) can be seen as a depth-first traversal of the rotation graph $R_{\mathcal{F}}$ that starts at vertex ν and that does *not* visit any vertex ν' that satisfies one of the following *termination conditions*:

$(t1)$ ν' is a distinguishing assignment for a clause c' that is already in $\mathcal{M}$, or

$(t2)$ ν' is not a distinguishing assignment, that is $|Unsat(\mathcal{F}, \nu')| > 1$.

We will refer to a vertex ν of $R_{\mathcal{F}}$ as *visited* by RMR, if the algorithm was invoked with ν as a parameter, either directly or recursively. For example, the vertices visited during the execution of RMR($\mathcal{F}, \mathcal{M}, \nu_1$) on the formula $\mathcal{F}$ from Examples 2 and 3 are: $\nu_1 = \{x_1, x_2, x_3\}$, $\nu_2 = \{\neg x_1, x_2, x_3\}$, $\nu_3 = \{x_1, \neg x_2, x_3\}$, and $\nu_4 = \{x_1, \neg x_2, \neg x_3\}$.

The conditions $(t1)$ and $(t2)$ guarantee that any execution of recursive model rotation visit at most $O(|\mathcal{F}|)$ vertices. At the same time, as demonstrated in the following example, the conditions inhibit the ability of the procedure to detect necessary clauses.

Example 4 *Let $\mathcal{F} = \{c_1, \ldots, c_6\}$, where*

$$c_1 = \{x_1, x_2\} \quad c_3 = \{\neg x_1, \neg x_3\} \quad c_5 = \{\neg x_2, \neg x_3\}$$
$$c_2 = \{x_3, x_4\} \quad c_4 = \{\neg x_1, \neg x_4\} \quad c_6 = \{\neg x_2, \neg x_4\}$$

Note that $\mathcal{F}$ is minimally unsatisfiable. Assume that the set $\mathcal{M}$ of known transition clauses is empty.

First, consider the execution of RMR($\mathcal{F}, \mathcal{M}, \nu_1$) where $\nu_1 = \{\neg x_1, \neg x_2, x_3, x_4\}$. We have $Unsat(\mathcal{F}, \nu_1) = \{c_1\}$. Since x_1 and x_2 are in $Var(c_1)$, RMR can attempt to flip the two variables, however $Unsat(\mathcal{F}, \nu_1|_{\neg x_1}) = \{c_3, c_4\}$ and $Unsat(\mathcal{F}, \nu_1|_{\neg x_2}) = \{c_5, c_6\}$. Thus, neither of the two assignments (note that they are neighbours of ν_1 in $R_{\mathcal{F}}$) are distinguishing, and RMR terminates without detecting any additional clauses due to the condition $(t2)$.

Second, consider the execution of RMR($\mathcal{F}, \mathcal{M}, \nu_2$) where $\nu_2 = \{\neg x_1, \neg x_2, x_3, \neg x_4\}$. We also have $Unsat(\mathcal{F}, \nu_2) = \{c_1\}$. We invite the reader to check that the assignment $\nu_3 = \{\neg x_1, x_2, \neg x_3, x_4\}$, which is the only distinguishing assignment for the clause c_6, will not be visited by RMR due to the condition $(t1)$, and so RMR will not detect the clause c_6.

In fact, in this example, at least one clause will be missed by RMR regardless of the initial distinguishing assignment.

Thus, a possible approach to enhancing the ability of RMR to detect additional transition clauses is to relax the termination conditions $(t1)$ and $(t2)$. In order to relax the condition $(t1)$ we allow the algorithm to visit a vertex ν even if it is a distinguishing vertex for a clause c that is known to be a transition clause. In order to guarantee the termination of the algorithm, we must ensure that the algorithm never re-visits any vertex — note that the condition $(t1)$ provides such guarantee implicitly. In order to relax the condition $(t2)$ we allow the algorithm to visit a vertex ν even if $|Unsat(\mathcal{F}, \nu)| > 1$. When this is the case, the **foreach** loop of RMR (Algorithm 4, lines 3-7) iterates over all variables in the clauses of $Unsat(\mathcal{F}, \nu)$.

While Example 4 demonstrates that the unrestricted version of RMR (i.e. with the conditions $(t1)$ and $(t2)$ relaxed as described above) has the potential to detect more transition clauses, a quick look at Example 3 and Fig. 4 reveals that in the worst case the algorithm may traverse all of the $2^{|Vars(\mathcal{F})|}$ vertices of the rotation graph. To control the worst-case computational complexity of the unrestricted algorithm we introduce two parameters – the *rotation depth* rd, $rd \geq 1$, and the *rotation width* rw, $rw \geq 1$ – and define the relaxed versions of the termination conditions $(t1)$ and $(t2)$ in the following way:

$(t1')$ ν' is a distinguishing assignment for a clause c' and the number of visited *distinct* distinguishing assignments for c' is greater than rd;

$(t2')$ $|Unsat(\mathcal{F}, \nu')| > rw$.

In other words, rotation depth is the maximum number of distinct distinguishing assignments allowed to be visited for any clause, while the rotation width controls the maximum number of unsatisfied clauses allowed in any visited assignment. Setting $rd = rw = 1$ gives the original termination conditions of RMR[4],

[4]Note that we tacitly assume that if $c \in \mathcal{M}$, then it has been visited by RMR at least once – in the context of Algorithm 3 this indeed is the case, since RMR is invoked for every newly discovered transition clause.

Algorithm 5: Extended Model Rotation with depth rd and width rw (EMR$_{rd,rw}$)

Input : $\mathcal{F}$ — an unsatisfiable CNF formula
: $\mathcal{M} \subseteq \mathcal{F}$ — a set of transition clauses of $\mathcal{F}$
: ν — a model of $\mathcal{F} \setminus \mathcal{S}$ for some $\mathcal{S} \subset \mathcal{F}$
Effect: $\mathcal{M}$ may contain additional transition clauses of $\mathcal{F}$

```
1  begin
2      foreach x ∈ Var(Unsat(F, ν)) do
3          ν' ← ν|_¬x
4          S ← Unsat(F, ν')
5          if |S| ≤ rw then                          // Check rotation width (condition (t2'))
               // Ensure that ν' was not visited, and check rotation depth (condition (t1'))
6              if ν' ∉ visited(S) and |visited(S)| < rd then
7                  if |S| = 1 then                   // Found a new transition clause
8                      M ← M ∪ S
9                  visited(S) ← visited(S) ∪ {ν'}    // ν' is visited
10                 EMR_{rd,rw}(F, M, ν')             // Recurse
11 end
```

while setting $rd = rw = \infty$ results in the unrestricted version of RMR.

The resulting algorithm parametrized by rd and rw, *extended model rotation*, or *EMR$_{rd,rw}$*, is presented in Algorithm 5. The algorithm maintains a global (i.e. saved between the invocations) datastructure $visited$ which, for each set $\mathcal{S} \subseteq \mathcal{F}$ with rw or less clauses, keeps the set of assignments ν such that $Unsat(\mathcal{F}, \nu) = \mathcal{S}$. For example, $visited(\{c\})$ contains up to rd distinguishing assignments for the clause c. EMR is invoked by the hybrid MUS extraction algorithm instead of RMR whenever a new transition clause is detected via a call to SAT solver (Algorithm 3, line 10).

Proposition 3 *The number of assignments visited by Algorithm 3 with RMR replaced by EMR$_{rd,rw}$, for a fixed finite rd and rw, is $O(rd \cdot m^{rw})$, where m is the number of clauses in the input formula.*

Proof. Let $\mathcal{F}$ be the unsatisfiable input formula. In the worst case the algorithm will visit rd distinct assignments for each of the subsets of $\mathcal{F}$ of size rw or less. The number of such subsets is bounded by $|\mathcal{F}|^{rw}$. □

Proposition 3 implies that using extended model rotation with large depth, and particularly with large width, on application benchmarks is not practical – for example, even for $rw = 2$ the algorithm might incur run-time and memory overhead of the order of $|\mathcal{F}|^2$. We evaluated the performance of the algorithm with various settings for the values of rd and rw, and present the results of the evaluation for $(rd = 5, rw = 1)$ and $(rd = 1, rw = 2)$ in Fig. 5. The plots compare the percentage of transition clauses detected by EMR vs that of RMR, and the impact of EMR on the run-time of the hybrid MUS extraction algorithm. We observe that, as expected, the increase in the rotation depth, and more so in the rotation width [5], allows EMR to detect more transition clauses (Fig. 5, top-left and bottom-left). However, even for $(rd = 5, rw = 1)$ this increase does not payoff in terms of run-time, and in fact appears to inhibit performance on some of the instances (Fig. 5, top-right). The situation is worse for the case $(rd = 1, rw = 2)$ (Fig. 5, bottom-right), where in fact the algorithm runs out of memory on 78 benchmark instances solved by HYB (with RMR).

On the theoretical note, it is unclear whether an unrestricted version of EMR is capable of detecting *all* transition clauses of a given unsatisfiable formula $\mathcal{F}$. This question boils down to answering whether for any formula $\mathcal{F}$ there is a traversal of the rotation graph $R_{\mathcal{F}}$ that visits at least one distinguishing assignment for each transition clause $c \in \mathcal{F}$. Based on our computational experiments we put forward the following conjecture.

Conjecture 1 *Let $\mathcal{F}$ be a minimally unsatisfiable CNF formula, and let $R_{\mathcal{F}}$ be the rotation graph of $\mathcal{F}$. Then, there exists a distinguishing assignment ν such that the traversal of $R_{\mathcal{F}}$ starting from ν visits at least one distinguishing assignment for each clause $c \in \mathcal{F}$.*

[5] Recall that RMR is equivalent to EMR with $(rd = 1, rw = 1)$.

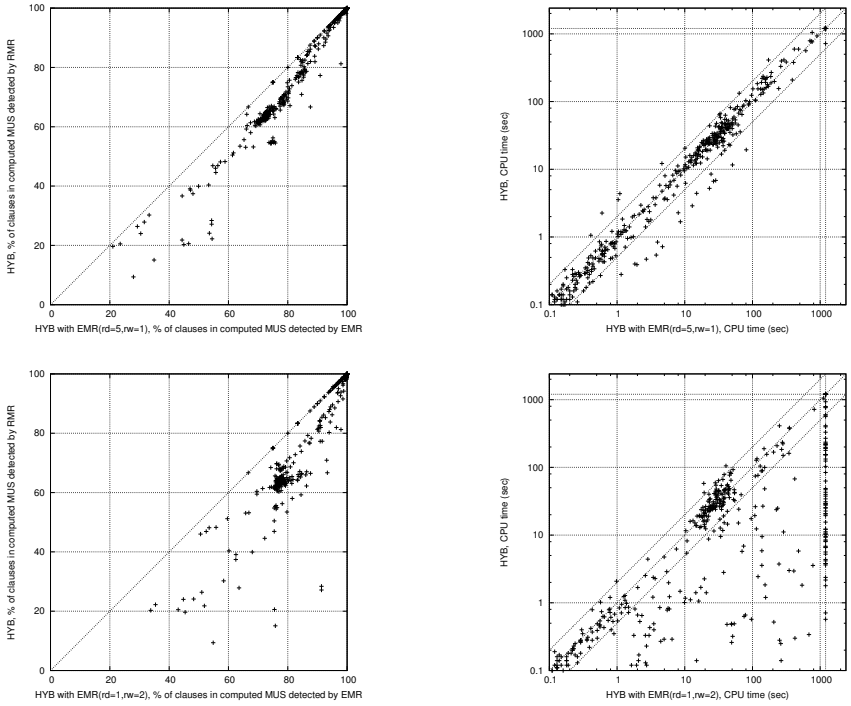

Fig. 5. HYB (with RMR) vs HYB with EMR with parameters settings $(rd = 5, rw = 1)$ (top row), and $(rd = 1, rw = 2)$ (bottom row). Left column: comparison of the percentage of clauses in the computed MUS that were detected by RMR vs EMR. Right column: comparison of CPU time (sec) of HYB with RMR vs EMR.

In other words, given a minimally unsatisfiable formula $\mathcal{F}$ and a "right" initial assignment, the unrestricted version of model rotation will discover all clauses of $\mathcal{F}$. We leave it as an open question to prove or refute the conjecture.

We now go back to the original version of RMR, and describe an additional technique that allows to use some of the information derived during the execution of the algorithm. This technique, called *clause reordering* is based on the following observation.

Proposition 4 *Let $\mathcal{F}$ be an unsatisfiable formula. Then, for any assignment ν the set $Unsat(\mathcal{F}, \nu)$ contains at least one clause from each of the MUSes of $\mathcal{F}$.*

Proof. If not, then the set $\mathcal{F} \setminus Unsat(\mathcal{F}, \nu)$ includes an MUS of $\mathcal{F}$, and so must be unsatisfiable. □

Proposition 4 justifies the following heuristic for selection of clauses in the hybrid MUS extraction algorithm: whenever RMR visits an assignment ν with $|Unsat(\mathcal{F}, \nu)| > 1$ (i.e. the test on line 5 of Algorithm 4 fails because ν' is not a distinguishing assignment), try to remove the clauses in $Unsat(\mathcal{F}, \nu)$ next (i.e. the clause c_i selected on line 5 of Algorithm 3 is selected from this set). The idea is that for instances with many MUSes chances are that the clauses from this set belong to *different* MUSes, and so among the next few calls to the SAT solver, the solver will return UNSAT and the clause set refinement will remove the clauses outside of the unsatisfiable core. We evaluated clause reordering empirically and present the results of the evaluation in Fig. 6. We observe that while clause reordering allows to solve 2 instances previously unsolved by HYB, in general the results are mixed. We conclude that there is no clear advantage in using the technique in the current form.

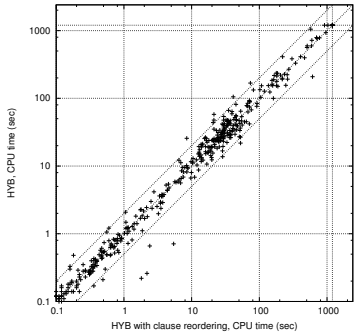

Fig. 6. Scatter plot comparing HYB with HYB with clause reordering in terms of CPU time (sec).

8. Related Work

To the best of our knowledge, Algorithm 2 was first presented in an earlier version of this paper [40]. Nevertheless, the use of relaxation variables for MUS extraction has been proposed in earlier work. For example, AMUSE [46] also uses relaxation variables. However, AMUSE does not compute an MUS, and identifies instead a reduced unsatisfiable subset. The use of relaxation variables has also been considered extensively in the enumeration of MUSes [35,37], and in the use of MaxSAT for MUS extraction [16]. Although the use of relaxation variables resembles the use of selector variables [44], 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 was also first presented in an earlier version of this paper [40], 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 [15,44]. 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 [51], in the context of a constructive algorithm requiring $\Theta(m \times k)$ calls to a SAT solver. In [51], 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. Model rotation was proposed in earlier versions of this paper [40,4]. Plain model rotation was proposed in [40] and recursive model rotation was proposed in [4]. This paper complements the study of model rotation by providing a detailed analysis of its impact in efficient MUS extraction, of possible extensions, and some of its limitations. Finally, the technique of including $\{\neg c_i\}$ in the CNF formula given to the SAT solver is standard in CNF redundancy checking [34], and was first used for MUS extraction in [51]. Our implementation follows this approach. Nevertheless, this paper proposes a new solution for integrating the redundancy removal technique and clause set refinement.

Recent work on MUS extraction also addressed non-clausal formulas [5] and group-oriented (or high-level) MUS extraction [44,48].

9. Conclusions

This paper details new algorithms for the efficient extraction of MUSes from unsatisfiable CNF formulas, first proposed in [40,4], and has a number of 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, more importantly, model rotation. These techniques 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. Moreover, although the proposed techniques are integrated in the new hybrid algorithm, they can be used with any MUS extraction algorithm. Among the techniques studied in this paper, model rotation is shown to be crucial for the practical efficiency of MUS extraction algorithms, and therefore is analyzed in greater detail. The techniques proposed in this paper (and related earlier work [40,4]) represent what seems to be the most effective organization of model rotation, and this paper provides insights on why this is the case. The resulting algorithm (*HYB*) outperforms publicly available MUS extraction tools. The performance gains often exceed one order of magnitude when compared with state of the art MUS extraction tools. In addition, algorithm *HYB* is shown to also outperform recent non-publicly available MUS extraction algorithms [44]. Finally, it is worth mentioning that the *HYB* algorithm as well as the implementation of several other MUS extraction algorithms is publicly available in the MUSer (MUS ExtractoR) software tool.

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. A number of research directions can be envisioned. MUSes find a wide range of practical applications in a number of domains (e.g. see [38]). One line of work is to adapt the *HYB* algorithm to other domains. A line of work recently investigated by other researchers is a tighter integration of the MUS extraction algorithm with the SAT solver. Some techniques developed in this paper could be further explored with this tighter integration. Finally, another line of research is to integrate in MUS extraction algorithms SAT solvers implementing the most recent SAT techniques, and evaluate their effectiveness for MUS extraction.

Acknowledgement.

This work is partially supported by SFI PI grant BEACON (09/ IN.1/I2618), FCT through grants ATTEST (CMU-PT/ELE/0009/2009), POLARIS (PTDC /EIA-CCO/123051/ 2010), ASPEN (PTDC /EIA-CCO/ 110921/2009), and by INESC-ID multiannual funding from the PIDDAC program funds.

References

[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–534, 1995.

[2] R. L. Ashenhurst. The decomposition of switching functions. In *Prooceedings of the International Symposium on the Theory of Swithing*, pages 74–116, 1957.

[3] 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.

[4] A. Belov and J. Marques-Silva. Accelerating MUS extraction with recursive model rotation. In *Formal Methods in Computer-Aided Design*, 2011. http://www.cs.utexas.edu/users/ragerdl/fmcad11/papers/74.pdf.

[5] A. Belov and J. Marques-Silva. Minimally unsatisfiable Boolean circuits. In *Theory and Applications of Satisfiability Testing*, pages 145–158, 2011.

[6] A. Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:75–97, 2008.

[7] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *DAC*, pages 317–320, 1999.

[8] R. Bruni. On exact selection of minimally unsatisfiable subformulae. *Ann. Math. Artif. Intell.*, 43(1):35–50, 2005.

[9] H. Chen and J. Marques-Silva. Improvements to satisfiability-based boolean function bi-decomposition. In *VLSI-SoC*, pages 142–147. IEEE, 2011.

[10] J. W. Chinneck and E. W. Dravnieks. Locating minimal infeasible constraint sets in linear programs. *INFORMS Journal on Computing*, 3(2):157–168, 1991.

[11] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[12] O. Cohen, M. Gordon, M. Lifshits, A. Nadel, and V. Ryvchin. Designers work less with quality formal equivalence checking. In *Proceedings of the Design and Verification Conference and Exhibition (DVCon '10)*, 2010.

[13] H. A. Curtis. *New approach to the design of switching circuits*. Princeton, N.J., 1962.

[14] J. L. de Siqueira N. and J.-F. Puget. Explanation-based generalisation of failures. In *European Conference on Artificial Intelligence*, pages 339–344, 1988.

[15] 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.

[16] 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.

[17] E. A. Emerson and A. P. Sistla, editors. *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*. Springer, 2000.

[18] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. Satisfiability solvers. In *Handbook of Knowledge Representation*, pages 89–134. Elsevier, 2008.

[19] É. Grégoire, B. Mazure, and C. Piette. Extracting MUSes. In *European Conference on Artificial Intelligence*, pages 387–391, August 2006.

[20] 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.

[21] É. Grégoire, B. Mazure, and C. Piette. Local-search extraction of MUSes. *Constraints*, 12(3):325–344, 2007.

[22] 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.

[23] 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.

[24] A. Gupta, M. Ganai, Z. Yang, and P. Ashar. Iterative abstraction using sat-based bmc with proof analysis. In *Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*, ICCAD '03, pages 416–, Washington, DC, USA, 2003. IEEE Computer Society.

[25] B. Han and S.-J. Lee. Deriving minimal conflict sets by cs-trees with mark set in diagnosis from first principles. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 29(2):281–286, 1999.

[26] F. Hemery, C. Lecoutre, L. Sais, and F. Boussemart. Extracting MUCs from constraint networks. In *European Conference on Artificial Intelligence*, pages 113–117, 2006.

[27] J. Huang. MUP: a minimal unsatisfiability prover. In *Asia South Pacific Design Automation*, pages 432–437, 2005.

[28] S.-Y. Huang and K.-T. Cheng. *Formal Equivalence Checking and Design DeBugging*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[29] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke. Word level predicate abstraction and refinement for verifying rtl verilog. In W. H. J. Jr., G. Martin, and A. B. Kahng, editors, *DAC*, pages 445–450. ACM, 2005.

[30] U. Junker. QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In *AAAI Conference on Artificial Intelligence*, pages 167–172, 2004.

[31] O. Kullmann. Lean clause-sets: generalizations of minimally unsatisfiable clause-sets. *Discrete Applied Mathematics*, 130(2):209–249, 2003.

[32] D. Le Berre and A. Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[33] R.-R. Lee, J.-H. R. Jiang, and W.-L. Hung. Bi-decomposing large boolean functions via interpolation and satisfiability solving. In L. Fix, editor, *DAC*, pages 636–641. ACM, 2008.

[34] P. Liberatore. Redundancy in logic I: CNF propositional formulae. *Artif. Intell.*, 163(2):203–232, 2005.

[35] M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008.

[36] 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.

[37] M. H. Liffiton and K. A. Sakallah. Generalizing core-guided Max-SAT. In *Theory and Applications of Satisfiability Testing*, pages 481–494, 2009.

[38] J. Marques-Silva. Computing minimally unsatisfiable subformulas: State of the art and future directions. *Journal of Multiple-Valued Logic and Soft Computing*. In Press.

[39] J. Marques-Silva. Minimal unsatisfiability: Models, algorithms and applications. In *International Symposium on Multiple-Valued Logic*, pages 9–14, 2010.

[40] J. Marques-Silva and I. Lynce. On improving MUS extraction algorithms. In *Theory and Applications of Satisfiability Testing*, pages 159–173, 2011.

[41] 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.

[42] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In H. Garavel and J. Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2003.

[43] A. Mishchenko, B. Steinbach, and M. A. Perkowski. An algorithm for bi-decomposition of logic functions. In *DAC*, pages 103–108. ACM, 2001.

[44] A. Nadel. Boosting minimal unsatisfiable core extraction. In *Formal Methods in Computer-Aided Design*, October 2010.

[45] A. Nadel. Boosting minimal unsatisfiable core extraction. In R. Bloem and N. Sharygina, editors, *FMCAD*, pages 221–229. IEEE, 2010.

[46] 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.

[47] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[48] V. Ryvchin and O. Strichman. Faster extraction of high-level minimal unsatisfiable cores. In *Theory and Applications of Satisfiability Testing*, pages 174–187, 2011.

[49] C. Sinz, A. Kaiser, and W. Küchlin. Formal methods for the validation of automotive product configuration data. *AI EDAM*, 17(1):75–97, 2003.

[50] P. J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, Haskell '03, pages 72–83, New York, NY, USA, 2003. ACM.

[51] H. van Maaren and S. Wieringa. Finding guaranteed MUSes fast. In *Theory and Applications of Satisfiability Testing*, pages 291–304, 2008.

[52] 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.