MUS Extraction using Clausal Proofs

Anton Belov[1], Marijn J. H. Heule[2] and Joao Marques-Silva[1,3]

[1] Complex and Adaptive Systems Laboratory, University College Dublin
[2] The University of Texas at Austin
[3] IST/INESC-ID, Technical University of Lisbon, Portugal

Abstract. Recent work introduced an effective method for extraction of reduced unsatisfiable cores of CNF formulas as a by-product of validation of clausal proofs emitted by conflict-driven clause learning SAT solvers. In this paper, we demonstrate that this method for *trimming* CNF formulas can also benefit state-of-the-art tools for the computation of a Minimal Unsatisfiable Subformula (MUS). Furthermore, we propose a number of techniques that improve the quality of trimming, and demonstrate a significant positive impact on the performance of MUS extractors from the improved trimming.

1 Introduction

Recent years has seen a significant progress in efficient extraction of a Minimal Unsatisfiable Subformula (MUS) from a CNF formula [1,2,3,4,5]. However, most of the formulas that can be tackled relatively easily by SAT solvers are still too hard for today's MUS extraction tools. In the context of MUS extraction, the term *trimming* refers to a preprocessing step, whereby the input formula is replaced with a smaller unsatisfiable core. Trimming is typically performed by a repeated invocation of a SAT solver starting from the input formula (e.g. [2,6,7]), but the technique seldom pays off in practice. This is evidenced by the fact that none the state-of-the-art MUS extractors `MUSer2` [8], `TarmoMUS` [9], and `HaifaMUC` [3], employ an explicit trimming step.

In this paper, we propose techniques to trim CNF formulas using clausal proofs [10]. A clausal proof is a sequence of clauses that includes the empty clause and that are entailed by the input formula. Clausal proofs can easily be emitted by SAT solvers and afterwards be used to extract an unsatisfiable core. Trimming based on clausal proofs can substantially reduce the size of CNF formulas, and a recent tool `DRUPtrim` [11] made this approach very efficient. Hence, as suggested in [11], it might be effective for the computation of MUSes.

We, first, confirm empirically the effectiveness of trimming using clausal proofs for MUS extraction. In addition, we present three techniques to strengthen MUS extraction tools via clausal proofs. These techniques are designed to both reduce the size of the trimmed formula further, and to compute a useful resolution graph — a crucial component of resolution-based MUS extractors. The first technique converts a clausal proof into a resolution graph. This conversion requires significantly less memory compared to computing the graph within a SAT solver. The second technique, called *layered trimming*, was developed to reduce the size of cores and the resolution graphs. This is achieved by solving a formula multiple times while increasing the bound of variable elimination [12,13]. Our third technique adds interaction between the trimming procedure and an MUS extractor. If the extractor gets stuck, it provides the current over-approximation of an MUS to a trimmer to obtain a new resolution graph. The process is repeated until a MUS is found. Experimental results with the MUS extractors `MUSer2` and `HaifaMUC` show that the proposed techniques can boost the performance of the extractors, particularly on hard benchmarks.

2 Preliminaries

We assume familiarity with propositional logic, its clausal fragment, and commonly used terminology of the area of SAT (cf. [14]). We focus on formulas in CNF (*formulas*, from hence on), which we treat as (finite) (multi-)sets of clauses. Given a formula F we denote the set of variables that occur in F by $Var(F)$. An *assignment* τ for F is a map $\tau : Var(F) \rightarrow \{0, 1\}$. Assignments are extended to formulas according to the semantics of classical propositional logic. If $\tau(F) = 1$, then τ is a *model* of F. If a formula F has (resp. does not have) a model, then F is *satisfiable* (resp. *unsatisfiable*). A clause C is *redundant* in F if $F \setminus \{C\} \equiv F$. Given two clauses $C_1 = (x \vee A)$ and $C_2 = (\bar{x} \vee B)$, the *resolution rule* infers the clause $C = (A \vee B)$, called the *resolvent* of C_1 and C_2 on x. We write $C = C_1 \otimes_x C_2$, and refer to C_1 and C_2 as the *antecedents* of C.

Resolution Graph. A *resolution graph* is a directed acyclic graph in which the leaf nodes (no incoming edges) represent the input clauses of a given formula. The remaining nodes in the resolution graph are resolvents. The incoming edges represent the antecedents of a resolvent. In case a node has more than two antecedents, it can be constructed using a sequence of resolution steps. The *size* of a resolution graph is the number of its edges.

Clausal Proofs. For a CNF formula F, *unit propagation* simplifies F based on unit clauses; that is, it repeats the following until fixpoint: if there is a unit clause $(l) \in F$, remove all clauses containing l and remove $\bar{l}$ from all clauses. We refer as a *lemma* to a clause that is logically implied by a given formula. A clausal proof [7] is a sequence of lemmas. To validate whether a lemma L in a clausal proof is indeed logically implied, one can check if unit propagation of $\bar{L}$, the assignment that falsifies all literals in L, results in a conflict. This check is also known as *reverse unit propagation* (RUP) [15]. Clausal proofs are significantly smaller than resolution proofs, and only minor modifications of a SAT solver are required to emit clausal proofs [11]. Clausal proofs can be converted into resolution graphs by marking clauses involved in a conflict as the antecedents of a lemma. Fig. 1 shows a CNF formula, a clausal proof and a resolution graph that can be obtained from it.

CNF formula

```
 p cnf 3 6
-2  3 0
 1  3 0
-1  2 0
-1 -2 0
 1 -2 0
 2 -3 0
```

clausal proof

```
   -2 0
d  -2 3 0
   -1 0
d  -1 2 0
    3 0
    0
```

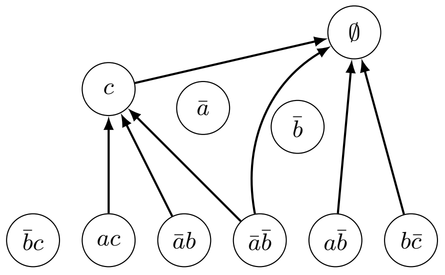

Fig. 1. An example CNF formula in DIMACS format and a clausal proof in DRUP format next to it. The files use three variables a, b, and c that are represented by `1`, `2`, and `3`, respectively. Each line in the proof refers to a clause addition step (no prefix) or a clause deletion step (`d` prefix). The `-2 3 0` represents the clause $(\bar{b} \vee c)$. On the right, a resolution graph that can be computed from the clausal proof. For all clauses in the graph only the literals are shown: $\bar{b}c$ means $(\bar{b} \vee c)$.

Minimal Unsatisfiability. A CNF formula F is *minimal unsatisfiable* if (i) F is unsatisfiable, and (ii) for any clause $C \in F$, the formula $F \setminus \{C\}$ is satisfiable. A CNF formula F' is a *minimal unsatisfiable subformula* (MUS) of a formula F if $F' \subseteq F$ and F' minimal unsatisfiable. While this paper focuses on MUS extraction on plain CNF formulas, the discussion can be extended to the computation of group-MUSes [16,17] without difficultly.

Practical MUS extraction algorithms are based on detection of *necessary* clauses. A clause C is *necessary* for F if F is unsatisfiable and $F \setminus \{C\}$ is satisfiable [18]. A basic *deletion*-based algorithm for MUS [6] extraction operates in the following manner. Starting from an unsatisfiable formula F, we repeat the following until fixpoint. Pick an unexamined clause $C \in F$ and solve $F \setminus \{C\}$. If the result is unsatisfiable, C is permanently removed from F, otherwise it is included in the computed MUS. The basic deletion algorithm with *clause-set refinement* and *model rotation* [19] is among the top performing algorithms for industrially relevant instances [20,2,3]. Additionally, reuse of lemmas and heuristic values between invocations of the SAT solver is crucial for performance [20]. Currently, the two prevalent approaches to achieving lemma reuse in the context of MUS extraction are the *assumption-based* and the *resolution-based* [17].

The assumption-based approach to MUS extraction relies on the incremental SAT solving paradigm [21]. Each clause C_i of the input CNF formula F is augmented with a fresh *assumption literal* a_i, and the modified formula is loaded once into an incremental SAT solver. To test a clause C_i for necessity the SAT solver is invoked under assumptions a_i and $\bar{a}_j$, for $j \neq i$. If the outcome is SAT, C_i is necessary for the given formula, and, as an optimization, can be *finalized* by adding a unit clause $(\bar{a}_i)$ to the SAT solver. If the outcome is UNSAT, C_i is "removed" by adding the a unit clause (a_i) to the SAT solver.

The resolution-based approach to MUS extraction relies on a modified SAT solver that constructs a resolution graph during search *explicitly*. The initial graph that represents a refutation of the input formula F is constructed when the formula is shown to be UNSAT. To test a clause $C \in F$ for necessity a resolution-based MUS extractor temporarily disables C and all of its descendant lemmas in the resolution graph, and invokes the SAT solver's search procedure on the remaining clauses. If the outcome is SAT, the lemmas are put back. In the case of the UNSAT outcome, the solver constructs a new resolution graph. A particularly important, in our context, feature of resolution-based MUS extractors is that the initial resolution graph can be provided as an input to the extractor, thus eliminating the cost of the first UNSAT call.

3 Trimming Strategies

Proofs of unsatisfiability can be used to *trim* a CNF formula F, i.e., remove the clauses from F that were not required to validate the proof. In this section we propose three new strategies to combine a trimming utility for clausal proofs with a MUS extraction tool. Spending a significant amount of time on trimming can improve the overall performance, particularly on hard formulas.

Trimming via Clausal Proofs. Resolution graphs are a crucial component of resolution-based MUS extraction tools. However, computing them while solving a given formula can be very costly, especially in terms of memory. Alternatively, one can compute a trimmed formula and a resolution graph by validating a clausal proof [11]. This approach requires significantly less memory.

We refer to a *trimmer* as a tool that, given a formula F and a clausal proof P, computes a trimmed formula F_{trim} and a resolution graph G_{res}. An example of such a tool is `DRUPtrim` [11]. Emitting clausal proofs is now supported by many state-of-the-art CDCL solvers. Consequently, we can compute trimmed formulas and resolution graphs efficiently using these solvers with a trimmer. The trimmed formulas are useful for assumption-based MUS extractors, while the resolution graphs are useful for resolution-based MUS extractors.

Fig. 2 shows the pseudo-code of the TrimExtract procedure that repeatedly trims a given formula F using a clausal proof that is returned by a SAT solver (the Solve procedure). The Trim procedure returns a trimmed formula and a resolution graph, which can be given to an extractor tool. The main heuristic in this loop deals with when to stop trimming and start extracting. Throughout our experiments we observed that it is best to switch to extraction if a trimmed formula is only a few percent smaller than the formula from the prior iteration.

```
    TrimExtract (formula $F$)
TE1   forever do
TE2     $\langle F_{\text{trim}}, G_{\text{res}} \rangle$ := Trim ($F$, Solve ($F$))        // trim using a clausal proof
TE3     if $|F_{\text{trim}}| \approx |F|$ then return Extract ($F_{\text{core}}, G_{\text{res}}$)     // switch to extractor
TE4     $F := F_{\text{trim}}$                          // $F_{\text{core}}$ becomes $F$ for the next iteration
```

Fig. 2. Pseudo-code of TrimExtract that combines a trimming and MUS extractor tool.

```
Trim (formula $F$, clausal proof)
  return ⟨trimmed $F$, resolution graph⟩

Solve (formula $F$)
  return clausal proof of solving $F$

Simplify (formula $F$)
  return ⟨simplified $F$, clausal proof⟩
```

```
LayeredTrim (formula $F$, iterations $k$)
  $P := \emptyset$          // start with empty proof
  $W := F$          // make a working copy
  for $i \in \{1, ..., k\}$ do
    $\langle W_{\text{simp}}, P_{\text{simp}} \rangle :=$ Simplify($W$)
    $\langle W, G \rangle :=$ Trim($W_{\text{simp}}$, Solve($W_{\text{simp}}$))
    $P := P \cup P_{\text{simp}}$
  return Trim($F, P \cup$ Solve($W$))
```

Fig. 3. The LayeredTrim procedure and the required subprocedures.

Layered Trimming. Most lemmas in resolution proofs have hundreds of antecedents [11]. This has a number of disadvantages. First, storing the graph requires a lot of memory. Second, in a resolution-based MUS extractor, high connectivity causes large parts of the proof being disabled during each SAT call. Third, high connectivity causes additional clauses to be brought into the core.

Our next strategy, called *layered trimming*, aims at reducing the size of the resolutions graphs by adding lemmas with only two antecedents using variable elimination (VE) [12,13]. VE replaces some clauses by redundant and irredundant clauses. Trimming the VE preprocessed formula will remove some redundant clauses, allowing more applications of VE. In general, the smaller the formula, the faster the solver and hence the shorter the proof. After each iteration of layered trimming, the number of vertices in the proof is similar. Yet more and more vertices are produced by VE (two antecedents), while fewer vertices are produced by CDCL solving (many antecedents). This reduces the connectivity.

The pseudo-code of LayeredTrim is shown in Fig. 3. We initialize a working formula W with the input formula F. In each iteration, W first gets simplified using the variable elimination procedure, resulting in the formula W_{simp}. The simplification steps are emitted as a, possibly partial, clausal proof P_{simp}, and we start to build a proof P which accumulates all of the simplification steps. Next we solve and trim W_{simp}, and take this trimmed version as our new working formula W. At each iteration proof P is extended by the new simplification steps. After k iterations, we merge the layered simplification proof P with the final proof returned by Solve. Finally, Trim will use this merged proof and the original formula to compute a core and resolution graph.

Iterative Trimming. Our third strategy, called *iterative trimming*, adds interaction between the trimming and the resolution-based MUS extraction tools. Two observations inspired iterative trimming. First, despite the substantial trimming, and despite the availability of the resolution graph at the beginning of MUS extraction, the extractor can get stuck while solving hard instances. This indicates that the current resolution graph might no longer be useful. Thus, we terminate the extractor and pass the current over-approximation of an MUS from the extractor to the trimming tool in order to obtain a new core and a new resolution graph. The MUS extractor is then invoked again on the new graph, and this iterative process continues until an MUS is computed. Second, while layered trimming can be very effective on hard formulas, it is quite significantly more expensive than (plain) trimming, and so for instances that are already easy for MUS extraction after plain trimming, the effort does not pay off. If a good overall performance of an MUS extractor is more important than scalability, we can postpone layered trimming until the extractor indicates that the formula is hard.

4 Empirical Study

To evaluate the impact of our trimming strategies, we implemented a Python-based framework, called `DMUSer`, on top of the following tools: `Glucose-3.0` [22] solver to emit clausal proofs in DRUP format and for simplification; `DRUPtrim` [11] to trim formulas and emit resolution graphs in TraceCheck format [23]; `MUSer2` [8], an assumption-based MUS extractor with `Glucose-3.0`; and `HaifaMUC` [3], a resolution-based MUS extractor, modified to support TraceCheck input[4].

The benchmark set consists of 295 instances from the MUS Competition 2011 and 60 instances from SAT Competition 2009[5] which `Glucose-3.0` could refute within 1 minute. We removed 31 instances from the MUS track — most of them were extremely easy — as some of the tools used by `DMUSer` produced errors.

All experiments were performed on 2 x Intel E5-2620 (2GHz) cluster nodes, with 1800 seconds CPU time and 4 GB memory limits per experiment. The reported CPU runtimes for trimming-based configurations include the runtime for both the trimming and the MUS extraction stages.

The results of our study are presented in plots in Table 1 and Fig. 4 and 5. We use the following abbreviations. `MUSer2` and `HaifaMUC` represent these MUS extractors running directly on the input instance, while the other configurations represent trimming followed (or interleaved) with MUS extraction: `Tr-` configurations use the original trimming of [11], `LTr-` configurations use the layered trimming, and `ITr-` use the iterative trimming; the `-M2` configurations perform MUS extraction on the trimmed CNF formula using `MUSer2`, while the `-HM` configurations compute MUSes using the TraceCheck version of `HaifaMUC` on the resolution graph of the trimmed formula.

[4] The source code of `DMUSer` and the benchmark set used for the evaluation are available from `https://bitbucket.org/anton_belov/dmuser`.

[5] The benchmarks are available via `http://satcompetition.org/`.

Table 1. The number of solved, timed- and memmed- out instances (out of 324), and the descriptive statistics of the CPU runtime (sec) of various configurations. The average is taken over the solved instances only, while the median is over all instances.

	`MUSer2`	`HaifaMUC`	`Tr-M2`	`Tr-HM`	`LTr-M2`	`LTr-HM`	`ITr-HM-A`	`ITr-HM-B`
Num. solved	250	258	257	266	273	277	**280**	276
Num.TO/MO	26/48	40/26	39/28	51/7	32/19	47/0	44/0	48/0
Med. CPU time	45.08	30.65	40.64	23.70	54.07	33.87	35.77	**23.16**
Avg. CPU time	97.58	110.09	102.95	102.03	162.62	108.52	117.07	112.12

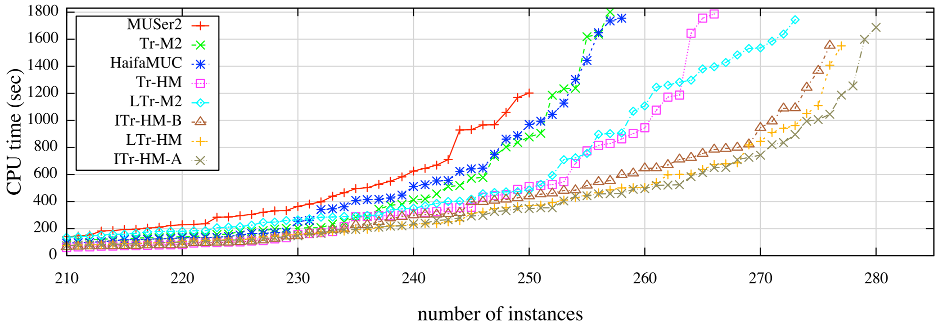

Fig. 4. Comparison of various trimming and MUS extraction techniques.

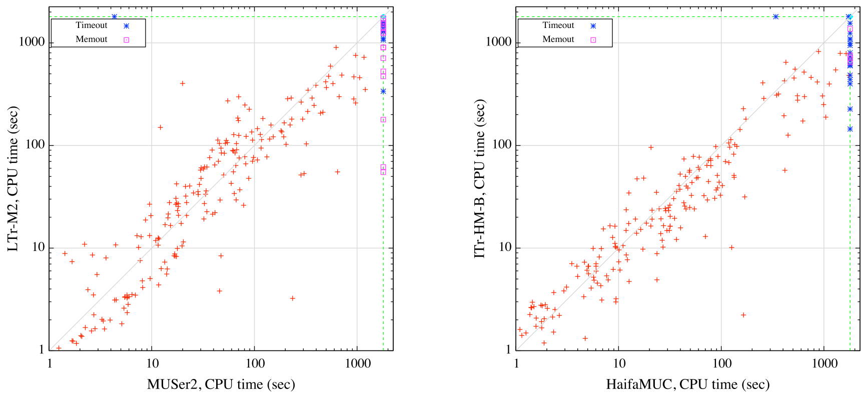

Fig. 5. Comparative performance of various trimming-based configurations against MUS extractors on non-trimmed formulas in terms of CPU runtime. Timeout of 1800 seconds is represented by the dashed (green) lines.

Impact of trimming. DRUP-based trimming for MUS extraction was suggested already in [11], but the impact has not been previously evaluated. Our results demonstrate that trimming is indeed an effective preprocessing technique for computing MUSes. The median reduction in the size of the input formula due to trimming is over 6x, with the average (resp. median) size of the trimmed formula being 1.5x (resp. 1.08x) of the size of the computed MUS. Comparing `MUSer2` vs. `Tr-M2` and `HaifaMUC` vs. `Tr-HM` on the cactus plot in Fig. 4 and in Table 1 we observe a notable performance improvement both with respect to `MUSer2` and to `HaifaMUC` (7 and 8 extra instances, respectively). Importantly, the MUS extractors run out of memory on fewer instances when executed on the trimmed formulas. Also, notice the decrease in median runtimes, indicating that trimming has an overall positive impact even on the relatively easy instances.

Impact of layered trimming. Our experimental data confirms the intuition that motivates the layered trimming technique: the lower connectivity in the resolution graph results in improved trimming and a lower memory consumption.

We did not use a fixed k for the iterations, but repeated the loop until the solving time no longer decreased. On average (resp. median) the size of the cores produced by the layered trimming is 1.42x (resp. 1.06x) smaller than the size of the cores produced by (plain) trimming, and constitutes a mere 1.04x (resp. 1.01x) of the size of the computed MUS. Layered trimming resulted in solving 16 (resp. 11) extra instances using `MUSer2` (resp. `HaifaMUC`) which is also clear from the cactus plot in Fig. 4. Notably, as seen in Table 1, `LTr-HM` had not ran out of memory on any instance – thanks to the smaller resolution graphs. The proposed technique, however, is not without drawbacks: observe the increase in the median runtime due to layered trimming, shown in Table 1. The scatter plot on the left of Fig. 5, that compares `MUSer2` vs. `LTr-M2`, gives some clues: layered trimming is a too heavy-weight technique for many of the easy instances. This might be undesirable in some applications.

Impact of iterative trimming. We experimented with two configurations of the iterative trimming algorithm. The configuration `ITr-HM-A` starts `HaifaMUC` with the resolution graph obtained from running the layered trimming approach. The algorithm aborts MUS extraction and returns to (plain) trimming when any SAT call takes too much time — 10 seconds, increasing linearly with every iteration. `ITr-HM-A` solves extra 3 instances, and performs notably better on the difficult instances, but the performance slightly decreases on easier instances.

The second configuration of iterative trimming, `ITr-HM-B`, is designed to alleviate the weaker performance of the proposed algorithms on the easier instances. This configuration starts the with resolution graph of plain trimming and aborts after some time (100 seconds in `ITr-HM-B`) and switches to `ITr-HM-A` starting with the resolution graph obtained by layered trimming. Although `ITr-HM-B` solves one less instance than `LTr-HM`, Fig. 5 (right) demonstrates that on most easier instances the configuration outperforms `HaifaMUC`, while still maintaining the significantly improved performance on the difficult instances.

5 Conclusions

We presented three trimming strategies to improve the performance of MUS extractors. Clausal proof based trimming is particularly useful when dealing with hard instances, but it can be costly on easy instances. Our layered trimming strategy reduces the memory consumption of resolution-based MUS extractors and can be useful in other applications that prefer low connectivity in resolution graphs. By applying the iterative trimming strategy, the performance of our MUS extraction tool is improved on both the easy and the hard instances.

Acknowledgements. The first and third authors are supported by SFI PI grant BEACON (09/IN.1/I2618), and by FCT grants ATTEST (CMU-PT/ELE/0009/2009) and POLARIS (PTDC/EIA-CCO/123051/2010). The second author is supported by DARPA contract number N66001-10-2-4087 and by the NSF under grant CCF-1153558. We thank Vadim Ryvchin for adding TraceCheck support to `HaifaMUC`.

References

1. Marques-Silva, J., Janota, M., Belov, A.: Minimal sets over monotone predicates in boolean formulae. In Sharygina, N., Veith, H., eds.: CAV. Volume 8044 of Lecture Notes in Computer Science., Springer (2013) 592–607
2. Belov, A., Lynce, I., Marques-Silva, J.: Towards efficient MUS extraction. AI Communications **25**(2) (2012) 97–116
3. Nadel, A., Ryvchin, V., Strichman, O.: Efficient MUS extraction with resolution. [24] 197–200
4. Audemard, G., Lagniez, J.M., Simon, L.: Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. [25] 309–317
5. Lagniez, J.M., Biere, A.: Factoring out assumptions to speed up MUS extraction. [25] 276–292
6. Dershowitz, N., Hanna, Z., Nadel, A.: A scalable algorithm for minimal unsatisfiable core extraction. In Biere, A., Gomes, C.P., eds.: SAT. Volume 4121 of LNCS. (2006) 36–41
7. Zhang, L., Malik, S.: Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In: DATE, IEEE Computer Society (2003) 10880–10885
8. Belov, A., Marques-Silva, J.: MUSer2: An efficient MUS extractor. Journal of Satisfiability **8** (2012) 123–128
9. Wieringa, S., Heljanko, K.: Asynchronous multi-core incremental SAT solving. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013). Number 7795 in Springer LNCS (2013)
10. Goldberg, E.I., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: DATE. (2003) 10886–10891
11. Heule, M.J.H., Hunt Jr., W.A., Wetzler, N.: Trimming while checking clausal proofs. [24] 181–188
12. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3) (1960) 201–215
13. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In Bacchus, F., Walsh, T., eds.: SAT. Volume 3569 of LNCS., Springer (2005) 61–75
14. Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press (2009)
15. Van Gelder, A.: Verifying RUP proofs of propositional unsatisfiability. In: ISAIM. (2008)
16. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reasoning **40**(1) (2008) 1–33
17. Nadel, A.: Boosting minimal unsatisfiable core extraction. In: FMCAD. (October 2010) 121–128
18. Kullmann, O., Lynce, I., Marques-Silva, J.: Categorisation of clauses in conjunctive normal forms: Minimally unsatisfiable sub-clause-sets and the lean kernel. In: SAT. (2006) 22–35
19. Marques-Silva, J., Lynce, I.: On improving MUS extraction algorithms. In: SAT. (2011) 159–173
20. Marques-Silva, J.: Minimal unsatisfiability: Models, algorithms and applications. In: ISMVL. (2010) 9–14

21. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. **89**(4) (2003) 543–560
22. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In Boutilier, C., ed.: IJCAI. (2009) 399–404
23. Jussila, T., Sinz, C., Biere, A.: Extended resolution proofs for symbolic SAT solving with quantification. In Biere, A., Gomes, C.P., eds.: Theory and Applications of Satisfiability Testing (SAT). Volume 4121 of LNCS., Springer (2006) 54–60
24. Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013. In: FMCAD, IEEE (2013)
25. Järvisalo, M., Van Gelder, A., eds.: Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings. Volume 7962 of Lecture Notes in Computer Science. Springer (2013)